Software Cartography and Code Orientation

— FINAL DRAFT, March 11, 2011 —

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Adrian Kuhn
von Dübendorf

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Acknowledgements

TO BE DONE

Abstract

Despite common belief, software engineers do not spend most time writing code. An approximate 50–90% of development time is spent on *code orientation*, that is navigation and understanding of source code. This may include reading of local source code and documentation, searching the internet for code examples and tutorials, but also seeking help of other developers.

In this dissertation, we are interested in how to support software engineers when they are facing technical questions that involve code navigation and code understanding.

In order to learn about the code orientation needs of software engineers we investigated how developers find answers to technical questions. In a qualitative user study, we found that software engineers use at least four kinds of cognitive clues for code orientation: lexical clues, social clues, episodic clues and spatial clues. While lexical clues are well supported by current tools, developers particularly struggle to resolve social, episodic and spatial clues.

Based on those findings we present a series of tools that provide first-class support for the code orientation clues that developers rely on. Common to the presented tools is that they tap on unconventional information found in the source code (such as identifier names, comments, code ownership and versioning information about a system's evolution) in order to provide developers with code orientation clues that would be out of their reach without tool support.

Among the code orientation strategies used by developers, spatial clues stand out for not having a first-class representation in the ecosystem of source code. Given the potential of spatial clues for code navigation, we investigated how to best represent software systems using spatial visualizations.

We introduce *Software Cartography*, a novel cartographic visualization of software systems that enables code orientation by spatial clues. The *software map* visualizations created by our approach offer a spatial representation of software systems based on lexical information found in the system's source code. Software maps are stable over time and can be used by individual developers as well as shared by a team. We implemented a prototype tool and evaluated it in a qualitative user study. We found that software maps are most helpful to explore search results and call hierarchies.

Contents

Chapter 1

Introduction

Despite common belief, software engineers do not spend most time writing code. An approximate 50–90% of development time is spent on code navigation and understanding [1, 123, 104]. This may include reading of local source code and documentation, searching the internet for tutorials and code examples, but also seeking help of other developers.

User studies have found that software engineers use at least four kinds of cognitive clues for code orientation: lexical clues, social clues, episodic clues and spatial clues [123, 104, 170, 70, 113]. Examples of code orientation by these clues are: searching for an identifier name (lexical clue), contacting a coworker with known expertise (social clue), referring to a design decision that was made years ago (episodic clue), or navigating to the location of a piece of code within a source text file (spatial clue).

In our research, we are interested in how to support software engineers when they are facing technical questions that involve code navigation and code understanding. We investigated how software engineers find answers to technical questions in order to learn about their code orientation strategies (Chapter 3). We found that the way towards an answer is typically split into two parts. In a first part, developers aim to turn their fuzzy initial clue into a concrete textual clue, for example by running a series of web searches and inspecting the results. Developers particularly struggle to resolve social, episodic and spatial clues during the first part. In a second part, they use that newly found textual clue to query resources on the internet or in local documentation. This two-step procedure might be an artifact of current tool limitations, that is developers aim to find a lexical clue first because other clues are harder to follow up, or it might be a general problem-solution strategy. In either case, developers need better tool support to follow up non-textual clues such as social, episodic and spatial clues.

Among the code orientation strategies used by developers, spatial clues stand out for not having a first-class representation in the ecosystem of source code. Spatial clues typically lack a first-class counterpart in source code since software systems do not have an inherent spatial extent. Nevertheless, we found that

developers do refer to source code by *ad hoc* spatial properties, such as clues regarding the position of a code artifact on-screen or within a source text file. Given the potential of spatial clues for code navigation, we investigated how to best represent software systems using spatial visualizatons.

In this dissertation we introduce *Software Cartography*, a novel cartographic visualization of software systems that enables code orientation by on-screen spatial clues (Chapter 6). The *software map* visualizations created by our approach offer a spatial representation of software systems based on lexical information found in the system's source code. Software maps are stable over time and can be used by individual developers as well as shared by a team.

We implemented a prototype tool and evaluated it in a qualitative user study. We found that software maps are most helpful to explore search results and call hierarchies (Chapter 7).

1.1 Types of Code Orientation

We introduce the term *code orientation* in order to refer to the navigation and understanding activities that happen while developers are looking for an answer to their technical question. Code orientation is thus an umbrella term for code navigation and code understanding, and includes activities such as reverse engineering, software exploration and program comprehension.

In the context of this work, we categorize code orientation activities according to the kind of cognitive clues being followed-up (lexical clues, social clues, episodic clues and spatial clues) and characterize orientation tasks according to their aim (refind, discovery and learning) and their reach (ranging from current working set to the entire internet).

The aim of code orientation tasks can be "refinding" information or source code that was encountered in the past, discovering new information or source code, or learning how to solve a problem by talking to people with expertise, by reading a book, or by getting advice from a mailing list or from a blog post.

The reach of code orientation tasks can range from the current working set to the entire internet. The current working set can be a single method or all open files. The local codebase can be the current project or the entire codebase of a developer's company, but also includes all local documentation and learning resources. Eventually the ultimate reach is accessing the internet. The internet offers two kinds of knowledge bases: on the one hand there are dedicated code repositories that only contain source code; on the other hand, there is an amazing amount of code examples that are contained in blog posts and websites. The latter are a typically more useful for code orientation since they have been selected through collaborative filtering, that is the authors of the embedding content carefully selected those examples for their usefulness for learning and copy-pasting.

The reach of code orientation tasks often correlates with their aim. For example, refinding tasks are typically targeted at the local codebase, with which the developers are familiar, whereas discovery and learning tasks are typically

targeted at the internet. But just as likely, parts of the local code based might be unknown to a developer and thus become the target of a discovery task, or it may be that a developer recalls having read about the solution on the internet and attempts to refind information at a global scale on the internet. Generally, the nature of information resources changes when moving from a local to global information source. Local information sources are typically limited, homogenous and authored by a small group of trusted people with known expertise. Global information sources are typically unlimited, heterogenous and authored by people with unknown expertise and unknown trustworthiness.

In the following we introduce the categorization of clues that developers use for code orientation. We motivate the use of orientation clues by developers, provide examples, shed light on underlying assumptions, and discuss current tool support and its shortcomings.

Orientation by Lexical Clues

We found that lexical clues are by far the most common clue used by software engineers for code orientation. Examples of lexical clues are recalling the name of an identifier, or using a keyword to search the web for documentation or code examples.

Developers rely on lexical clues because they assume that names are meaningful, that there is a meaning to names and that names have been meaningfully chosen. Lexical clues are often pointers to lexical information found in source code, that is identifier names or the vocabulary of comments. However, lexical information is not limited to source code but also found in emails, bug reports and web sites. For example, following up a lexical clue might guide the developer to a wikipedia page that contains the algorithm he's looking for.

Simple keyword search and regular expressions are of great help to follow up lexical clues. However, a major problem with lexical clues is that developers have to guess how other people name things. (It is often said that there are two hard problems in software engineering, caching and naming.) Using information retrieval and natural language processing can help to go beyond the limitation of keyword-based approaches.

Orientation by Social Clues

Social clues are often not considered part of program comprehension, yet they are most helpful for discovery and learning. Coworkers are the most frequent source of information used by developers [104]. Examples of social clues are asking coworkers for help, or posting a question to a discussion board on the web where professionals share their expertise.

Developers rely on social clues because they assume that people have expertise, in particular that other people have more or different expertise so they can be of help to find answers. Typically, the knowledge in the mind of team members is more accurate than documentation. Social clues are often pointers to other persons from the developer's personal network, either a co-worker or a

friend. However, social clues are not limited to the personal network but can also be pointers to mailing lists and other expert groups that are ready to share their expertise online on the internet.

Support for code orientation by social clues is typically not present in development tools. The current state of the art is that developers have to recall the name of a person with expertise, which basically boils down to a lexical clue that has to be used as a proxy. Using techniques and ideas drawn from social media can help address these limitations and may provide access to social clues that are beyond the reach of the developer's personal network.

Orientation by Episodic Clues

Episodic clues are most helpful to refind source code that has been written or used in the past, since as humans we have strong episodic memories. Episodic clues are typically tied to personal memory, as for example in recalling the first-hand experience of a conference talk or of a pair programming session.

Developers rely on episodic clues because they assume that their knowledge in the past has been more accurate than their current knowledge. That is a valid assumption because as humans our episodic memory works far better than our structural memory. Typically developers recall "that" they knew the answer before, but not "what" exactly constituted the answer.

Episodic clues are often pointers to past interaction with books, mailing lists and other people, but may also be pointers to past snapshots of the current or a related software system. Information related to episodic clues is often stored in external databases, such as version control repositories and mailing list archives, and thus not integrated with development tools. Using data mining techniques and embedding the results in a story-telling visualization can help to address this limitation and may provide access to episodic clues that are beyond the reach of a single developer's personal experience.

Orientation by Spatial Clues

Spatial clues may help to reduce the cognitive load of orientation in a hyperlinked document space, such as software systems. We found that spatial clues can be of three kinds, either they are structural or conceptual as in recalling which class or architectural layer some functionality belongs to, or they are true spatial clues as in recalling the on-screen or within-text-file position of a given function.

Developers rely on spatial clues because as humans they have strong spatial capabilities. The spatial capabilities of our brain are impressive. We found in a user study that developers form a spatially meaningful internal mental model of software systems (Chapter 7) even though the external representation of source code has no inherent spatial dimension.

Current support for code orientation by spatial clues in development tools is *ad hoc* at best. The system is presented as a tree of alphabetically ordered files and inside a file the source code is linearized as a text file. While this

might accidentally help the developer to refind some code by a spatial clue, support for discovering yet unknown source code by spatial clues is limited. Providing developers with a cartographic visualization so they can use spatial on-screen clues for navigation and understanding of code may help to go beyond the limitation of source code's missing spatial extent.

1.2 Thesis Statement

We state our thesis as follows

> To support software engineers in code navigation and understanding we need development tools that provide first-class support for the code orientation clues that developers rely on. We need to tap unconventional information found in the source code in order to provide developers with code orientation clues that would be out of their reach without tool support.

1.3 Contributions

We present the following contributions that explore the support of code orientation clues by development tools. Common to all these contributions is that they tap unconventional information found in source code and thus provide developers with code orientation clues that would otherwise be out of their reach. For example, in Chapter 8 we present a story-telling visualization of a system's history that enables new hires to draw on episodic memories that they would not have access to otherwise.

Each of these contributions has been published as one or more peer-reviewed publications at international conferences or in international journals. For each contribution we categorize aim and reach of the provided code orientation clues, through which sources of information these clues are established, and how developers may query the development environment for those clues.

Software Cartography

Aim	Code orientation in general.
Reach	Local codebase and possibly history of a system.
Clues	Spatial (established through lexical and structural information).
Query	Visual analytics of a cartographic visualization.

Current tool support for code orientation by spatial clues is *ad hoc* at best, most striking being the lack of spatial on-screen representations of source code. Without such a representation, developer are barely able to draw on the strong spatial capability of the human brain.

We present *Software Cartography*, an approach that provides a novel cartographic on-screen visualization such that developers can start using spatial clues for code orientation. Since software has no inherent spatial dimenions, we use lexical and structural information found in the source code to establish a spatial layout of the local code base. The generated *software maps* are stable over time and can be shared among members of a team to establish a common mental model of the system. We implemented our approach in a prototype and evaluated it in a user study. We found that it is most helpful for spatially exploring search results and call hierarchies.

For a brief introduction please refer to the next section, and for more in-depth information please refer to Chapter 6 and Chapter 7. The work on software cartography has been published as two peer-reviewed conference papers [115, 112] one of which has been extended into a peer-reviewed journal paper [111]. The CODEMAP prototype and its evaluation in a user study have been realized with the support of David Erni and Peter Loretan and is featured as part of their Master's thesis [66, 129].

Lexical Clustering

Aim	Refinding and discovering topics.
Reach	Local codebase of a system.
Clues	Lexical (established through lexical information).
Query	Visual analytics and fuzzy keyword search.

Keyword matching and regular expressions are powerful means for code orientation by lexical clues. However, current tool support fails to meet the developer needs when they are following up on a fuzzy lexical clue.

We present an approach to model a system's lexical information in a statistical text model that resolves synonymy and polysemy with unsupervised learning. We use the statistical text model to cluster the parts of a system by topic, and visualize the topics using correlation matrices and *distribution maps*, a specialized visualization that illustrates the distribution of topics over the packaging of a system. We implemented the approach in a prototype and evaluated its application.

For more information please refer to Chapter 4. The work on lexical clustering (formerly also known as "semantic clustering") has been published as a peer-reviewed conference paper [109] which has been extended as my Master's thesis [107] and as a peer-reviewed journal paper [110].

Code Summarization

When developers encounter a piece of source code for the first time, they are typically not presented with a high-level summary of the code's topics. We present an approach to summarize a piece of code as a word cloud, consisting of the statistically most significant terms that set this part of system apart

Aim	Refinding and discovering topics.
Reach	Part of a system's codebase or history.
Clues	Lexical and episodic (established through lexical information).
Query	Visual analytics of word clouds.

from the rest. The same approach can be used to compare two systems or two versions of the same system. Presenting those word clouds to the developer helps to lexically query the topics, as well to recover and tell the story of a system's history and thus enabling developers to draw from episodic memories that they possibly never experienced first-hand. We implemented the approach in a prototype and evaluated its application.

For more information please refer to Chapter 5. The work on code summarization has been published as a peer-reviewed conference paper [108].

Stories of Collaboration

Aim	Learning about team collaboration.
Reach	Team of a project's local codebase.
Clues	Episodic (established through social and historical information).
Query	Visual analytics of a story-telling visualization.

Episodic clues are of great help to developers when having to find their way through a system, however episodic memory is only available to those developer who know the system's history first-hand. We present an approach to recover and tell a system's history as a story-telling visualization. Our approach uses social and historical information taken from the version control system to establish an episodic visualization of the system's history. Both new hires and seasoned team members can use this visualization to learn about episodes from the system's history in order to take better technical decisions when working with the system in the future. We implemented the approach in a prototype and evaluated its application.

For more information please refer to Chapter 5. The work on visualizing code ownership is part of Mauricio Seeberger's Master's thesis [166] and has been published as a peer-reviewed conference paper [75] that was co-authored by Tudor Gîrba and myself.

Discovery of Experts

Aim	Discovery of experts.
Reach	Experts who committed to the local codebase.
Clues	Social (established through lexical and historical information).
Query	Fuzzy problem description given as natural language text.

Given current tool support, social clues have to be followed up through the lexical proxy of a person's name. We present an approach to discover experts without having to know their names. Given a problem description, such as a work item or a bug report, we provide automated means of linking the person with the expertise on that matter. In order to model the developer's expertise we use lexical information found in their contributions to the version control system. We implemented the approach in a prototype and evaluated it against a benchmark that consists of bug-report assignments.

The work on discovery of experts has been realized with Dominique Matter, and has been published as a peer-reviewed conference paper [141] as well as in Master's thesis [140].

Credibility of Code Search

Aim	Discovery of trustworthy projects.
Reach	Open-source projects on the internet.
Clues	Episodic (established through social and historical information).
Query	Name of an open-source project.

Searching for code examples or libraries on the internet is a common programming task. In interviews with developers, we have found that credibility is one of the major issues when copying source code from an external and thus untrusted source such as the internet (Chapter 3). We present an approach to automatically assesses the trustworthiness of open-source projects based on the credibility of their authors. Our approach infers the trustworthiness of unknown projects from the trustworthiness of well-known projects, if they have common contributors. We implemented the approach in a prototype and evaluated it against a benchmark in bug-report assignment .

The work on discovery of experts has been realized by my student Florian Gysin, and has been published as a peer-reviewed workshop paper [81] that he first-authored as well as his Bachelor's thesis [80], in addition his work won the ACM Student Competition award 2010 [79].

1.4 Software Cartography in a Nutshell

Current tool support for code orientation by spatial clues is *ad hoc* at best, most striking being the lack of spatial on-screen representations of source code. Without such a representation, developers are barely able to draw on the strong spatial capability of the human brain. With Software Cartography we aim to address this limitation. We provide a cartographic on-screen visualization that software engineers can start using to obtain spatial clues for code orientation. Since software has no inherent spatial structure, we use lexical and structural information found in the source code to establish a spatial layout.

Software Cartography embeds a cartographic visualization of the current working set in the development environment (IDE) of software engineers. The

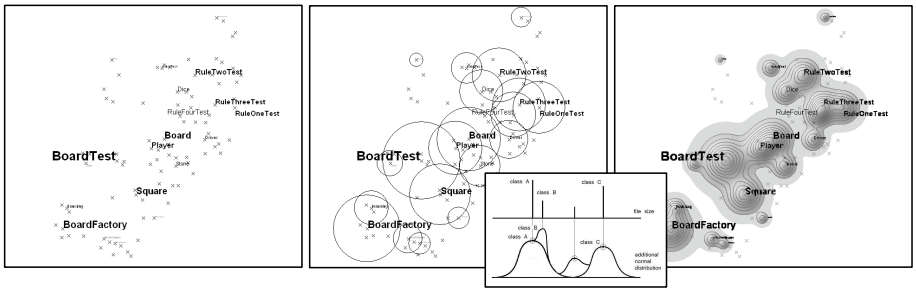


Figure 1.1: *Construction steps of a software map, from left to right: 1) 2-dimensional embedding of files on the visualization pane; 2.a) circles around each file's location, based on class size in KLOC; 2.b) each file contributes a Gaussian shaped basis function to the elevation model according to its KLOC size; the contributions of all files are summed up; 3) fully rendered map with hill-shading, contour lines, and filename labels.*

generated *software map* visualizations are stable over time and can be shared among members of a team to establish a common mental model of the system. Software maps are most useful when they support as many development tasks as possible with spatial clues. Therefore we integrated Software Cartography in the IDE so that a map of the software system may always be present and may thus support as many development tasks as possible.

The general approach of Software Cartography, as illustrated in Figure 1.1, 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 use Multidimensional Scaling (MDS) [33] to map the term-frequency histograms onto the 2D visualization pane. This preserves the lexical co-relation of source files as well as possible.

3. We use cartographic visualization techniques to render an aesthetically appealing landscape.

We implemented the approach in CODEMAP, a prototype plug-in for Eclipse that is available under an open-source license. The most recent version[1] of the plug-in supports the following tasks:

- Navigation within a software system, be it for development or analysis. CODEMAP is integrated with the package explorer and editor of Eclipse. The selection in the package explorer and the selection on the map are

[1] http://scg.unibe.ch/codemap

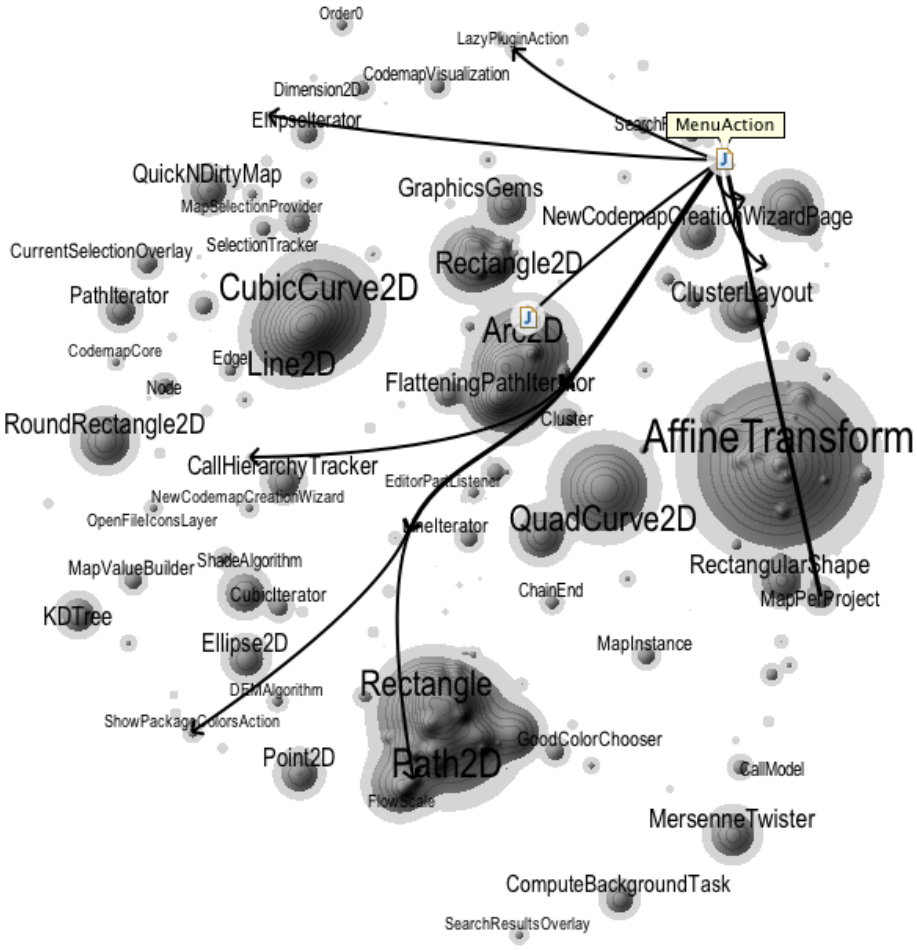


Figure 1.2: *Thematic codemap of a software system, here the* Codemap *tool itself is shown. Arrow edges show outgoing calls from the* `#getSettingOrDefault` *method in the* `MenuAction` *class, which is currently active in the editor and thus marked with a speech-balloon label.*

linked. Open files are marked with an icon on the map. Double clicking on the map opens the closest file in the editor. When using heat map mode, recently visited classes are highlighted on the map.

- Comparing software metrics to each other, for example to compare bug density with code coverage. The map displays search results, compiler errors, and (given the Eclemma plug-in is installed) test coverage information. More information can be added through an plug-in extension point.

- Social awareness of collaboration in the development team. CODEMAP can connect two or more Eclipse instances to show open files of other developers. Colored icons are used to show the currently open files of all developers. Icons are colored by user and updated in real time.

- Understand a software system's domain. The layout of CODEMAP is based on clustering software by topic [110], as it has been shown that, over time, the lexicon of source code is more stable than its structure [8]. Labels on the map are not limited to class names, but include automatically retrieved keywords and topics.

- Exploring a system during reverse engineering. CODEMAP is integrated with Eclipse's structural navigation features, such as search for callers, implementers, and references. Arrows are shown for search results. We apply the FLOW MAP algorithm [154] to avoid visual clutter by merging parallel arrow edges. Figure 1.2 shows the result of searching for calls to the `#getSettingOrDefault` method in the `MenuAction` class.

1.5 Outline

The dissertation is structured as follows

Chapter 2 discusses related work. We present various user studies and solutions to code orientation and analyse the shortcomings in the context of each orientation clue.

Chapter 3 presents a user study that looks at how developers find answers to technical questions and discusses code orientation by cognitive clues.

Chapter 4 presents an approach for clustering and summarizing of software systems using lexical information found in source code. The approach is implemented in the HAPAX tool.

Chapter 5 presents an approach that uses lexical information found in source code to summarize parts of a system, the whole system, or even the system's entire evolution. The approach is implemented in the EVOCLOUDS tool.

Chapter 6 introduces *Software Cartography*, an approach to establish a cartographic visualization that facilitates spatial code orientation by individuals or teams. The approach is implemented in the CODEMAP tool.

Chapter 7 reports on a qualitative user study that evaluates the prototype implementation of the Software Cartography approach presented above.

Chapter 8 presents an approach for addressing the episodic memory of developers by providing them with a visualization that tells the story of the team collaboration as recorded by the version control system. The approach is implemented in the CHRONIA tool.

Chapter 9 presents an approach that uses lexical information found in contributions that developers shared with open source systems to build a recommendation model for bug reports. The approach is implemented in the DEVLECT tool.

Chapter 10 presents an approach that uses cross-project collaboration of developers in open source projects to estimate the credibility of code search results. The approach is implemented in the BENDER tool.

Chapter 11 concludes the dissertation and outlines future work.

Chapter 2

Related Work

Investigation of human factors is a rather recent trend in the field of software engineering research. While there have always been papers that focused on human factors, a coherent body of research is yet to emerge. Researchers are starting to investigate the user needs of software engineers in qualitative studies, collecting for example the most frequent information needs of developers. Also, researchers in the field of development tool building are starting to evaluate the usability of their prototypes rather than just giving proof of a prototype's technical correctness.

At the same time, development tool building in industry is going through a similar transition. While it used to be that software engineers built their own tools (as often still is the case) we can observe a professionalization of development tool building as well as a change in the demographics of developers. Development tools are built by specialized engineers whereas the demographics of developers is growing to include more and more end-user programmers who lack the skills to build their own tools. In particular the store-based distribution models for mobile applications opened up a new market of end-user developers with very specific needs, but without the skills to build their own development tools. Independent of whether the assumption that development tool builders inherently know which tool to build because they address their own needs ever used to be valid or not—these days it certainly does not apply anymore: development tool building is about to become a specialized profession which opens up new and exciting research opportunities.

Related work on development tool building is basically found in two distinct bodies of research literature. On the one hand, there is the solution-driven work published at venues of the software maintenance community (such as ICPC[1], ICSM[2] and WCRE[3]). On the other hand, there is problem-oriented work published at venues of the human-computer interaction community (such as CHI[4],

[1] International Conference on Program Comprehension
[2] International Conference on Software Maintenance
[3] Working Conference on Reverse Engineering
[4] International Conference on Human-Computer Interaction

CSCW[5] and VLHCC[6]) and, as of recently, also the software engineering community (most prominently ICSE[7] with its CHASE[8] workshop).

The solution-driven literature typically reports on prototypes of external development tools that exploit novel technical approaches. Evaluation is done by applying the prototype on a small number of cases, typically the codebase of one or more open-source projects. This is done as proof-of-concept that it is feasible to realize the proposed technical approach and that the prototype implementation is technically correct. The user needs addressed by this kind work are typically not grounded in the findings of user studies, but rather drawn from the personal development experience of the researchers themselves. In recent years, some researchers in that field started to evaluate their prototypes using controlled experiments. Controlled experiments have been developed in psychology to study phenomena that are easy to isolate in the lab, such as basic human behavior. Isolating software engineering tasks in a lab situation is a daunting, often near impossible endeavour given the inherent complexity of software engineering and how little understood it still is. Using quantitative user studies is common practice to evaluate end-user applications, as are development tools, in industry.

The problem-oriented literature typically reports on surveys and interviews with professional developers and sometimes presents a tool prototype that addresses a specific user need identified by the initial user study. The results of the user studies are typically distilled as lists of common actions, questions or problems that document the most common or the most frequent user needs of software engineers. Evaluation of prototypes is done by qualitative user studies, that is again interviewing developers that used the prototype tool and thus showing that the tool in fact addresses the user need that it was built for. Some prototypes are evaluated using quantitative studies, typically not using controlled experiments but rather measuring frequency, correctness or performance of certain actions that are related to the addressed user need.

The remainder of this chapter is structured as follows. First, related work on developer needs is discussed. Then, tool prototypes are discussed, loosely grouped by the four categories of code orientation clues, that is lexical, social, episodic and spatial. Where applicable, literature on related information sources, such as for example lexical and social information found in source code, is discussed as well.

2.1 User Needs

The literature on developer needs is split into two bodies of work. More recent studies are descriptive and based on qualitative user studies whereas older work from the eighties and nineties tends to be theoretical and is typically drawn on

[5] International Conference on Computer-Supported Collaborative Work
[6] Symposium on Visual Languages and Human-Centric Computing
[7] International Conference on Software Engineering
[8] Workshop on Cooperative- and Human-Aspects in Software Engineering

personal experience rather than grounded in empirical studies.

Latoza, Venolia and Deline [123] studied the work habits of software engineers in two surveys and a series of interviews. They investigated how much time developers spend on code related activities and which are the most serious problems and questions that developers face when working with source code. They found that developers spend an almost equal amount of time (that is each about 10–15% of development time) on editing, writing, designing and understanding code, but also on communication with other developers and other activities, such as refactoring. They found a negative correlation between working on new features (36% of development time) and communication, which suggests that developers working on new features need less information from their team mates. Whereas developers working on bug fixes (49% of the development time) and maintenance (15% of development time) spend more time following up social clues, that is communication with team mates and their social network.

With regard to problems that developers face, they found that understanding the rationale behind a piece of code is the biggest problem for developers (reported by 66% of participants). When trying to understand a piece of code, developers turn first to the piece of code itself, and when that fails, follow up social clues to their personal network. Also among the top problems that developers face when working with source code are: being aware of changes that happen somewhere else (61%), understanding the history of a piece of code (51%), understanding who owns a piece of code (50%), and finding the right person to talk to (39%).

Ko, Deline and Venolia [104] studied the day-to-day information needs of software engineers by observing developers and transcribing their activities in 90-minute sessions. They identified twenty-one information types and catalogued the outcome and source when each type of information was sought. The most frequently sought information was awareness about artifacts and coworkers. The most often deferred searches included knowledge about design and program behavior, such as why code was written in a particular way, what a program was supposed to do, and the cause of a program state. Developers often had to defer tasks because the only source of knowledge was unavailable coworkers.

The identified programmer questions span seven categories of tasks: writing code, submitting a change, triaging bugs, reproducing a failure, understanding execution behavior, reasoning about design, and maintaining awareness. The most frequently information needs were: did I make any mistakes in my own code? what have my coworkers been doing? what caused this program state? in what situations does this failure occur? what is this program supposed to do? Their ranking might be biased though as most study participants were in a bug fixing phase.

Coworkers were the most frequent source of information, accessed at least once for 13 of the 21 information needs. The information needs where coworkers were most often consulted were episodic design knowledge and about execution behavior. Developers consulted coworkers because in most cases design knowledge was only in the coworker's mind. Even when design documentation was

available developers still turned to coworkers when they questioned the accuracy of the documentation.

Sillito, Murphy and De Volder [170] studied which questions software engineers ask when evolving a code base. They conducted two qualitative studies, one study involving newcomers and the other involving professional developers. Based on the studies they collected and categorized 44 questions that developers ask and how they find answers to these questions. They categorized the questions as follows: finding initial focus points (for example, which type represents this domain concept?), building on those points (for example, which types is this type part of?), understanding a subgraph of the system (for example, what is the behavior these types provide together?), and questions comparing groups of subgraphs (for example, what is the mapping between these user interface types and these domain model types?). They list the different features of the development environment that were used by programmers to answer these questions (such as search by keyword or using the debugger to verify a hypothesis about the program's behavior) and report that there were also times when no tool could provide direct assistance. Alas the study does not report on the use of other resources to answer questions, such as turning to the social network of programmers or using web search to find answers on the internet.

Fritz and Murphy [70] ran a series of interviews with software engineers and identified 78 questions that developers ask that are hard to answer. In their interviews, they focused on the variety and richness of questions rather than on their frequency. They grouped the questions by people-specific questions, code-specific questions concerning code change and ownership, question regarding the progress of work items, broken builds and test cases, as well as questions that refer to information found on the internet. The questions span eight domains of information: source code, change sets, teams, work items, websites and wiki pages, comments on work items, exception stack traces, and test cases. Because most questions require information from more than one domain of information, they present an information fragment model and a prototype tool that allows developers to combine information from different sources. They report that 94% of developers are able to easily answer selected question using their prototype tool.

Storey, Fracchia and Müller [175] propose a series of cognitive features that should be considered when designing software exploration (that is code orientation) tools. They propose that tools should: enhance bottom-up comprehension, enhance top-down comprehension, integrate bottom-up and top-down approaches, facilitate navigation, provide orientation clues, and reduce disorientation effects. They present SHriMP, a tool prototype that addresses all these issues and report on an evaluation with user studies.

The list of tools features as proposed by Storey *et al.* is based on program comprehension models taken from theoretical literature on program comprehension models. The bottom-up program comprehension model by Shneiderman and Pennington proposes that understanding is built by reading source code and then mentally chunking or grouping those statements into higher-level abstractions [169, 153]. The top-down program comprehension model by Brooks,

Soloway and Ehrlich proposes that developers understand a complete program by reconstructing knowledge about the domain of the program and mapping that to the actual code [37, 173]. The knowledge-based program comprehension model by Letovsky views developers as opportunistic processors capable of exploiting both bottom-up and top-down clues [124]. The systematic and as-needed program comprehension models by Littman and Soloway is based on the observation that developers either read all code in detail to gain a global understanding or that they focus only on the code related to their current task [128]. Von Mayrhauser and Vans synthesize Soloway's top-down model with Pennington's model in an integrated model of program comprehension: the combined model consists of a top-down comprehension model of the domain, a bottom-up comprehension model of the source code and a situation-based comprehension model of the execution behavior, plus a knowledge base of the programmers knowledge [189].

Common to all these comprehension models is that they consider program comprehensions as a task performed by single engineers who are typically considered to be new to a system, rather than as a collaborative activity of engineers that have knowledge both about the parts of the system they own as well as knowledge about who owns other parts of the system. Social and episodic clues are thus typically not included in these theoretical models.

When this research was published internet-scale search has not been available to developers. As we show in our user study in Chapter 3 the internet has since become the prime source of answers for developers when they are lost in third-party code. Theories such as Pirolli's *information foraging* theory [156, 155] might thus be more appropriate to model the program comprehension strategies of today's software engineers. Information foraging draws an analogy between human behavior when searching for information and the foraging mechanisms that evolved to help our ancestors find food. It has been proposed to use this analogy to better support information needs in user interfaces.

Also, to our best knowledge, there is no empirical research on generational differences in software engineering, except a brief discussion in Andrew Hunt's "Pragmatic Thinking and Learning" [95]. There might be generational differences that impact the application of findings from eighties on today's generation of software engineers in particular given the adoption of agile methodologies by the younger generation.

2.2 Lexical Information

Using data mining to exploit lexical information found in source code has received quite some attention in recent years. Publications on this topic are typically solution-driven and apply information retrieval algorithms on source code that are taken from work on natural language text. Latent semantic indexing [49] and, more recently, latent dirichlet analysis [18] have been adopted by the software maintenance community.

There have been attempts to apply ontologies to the lexical information

found in source code. However, based on communication with other researchers and personal experience, there have been no successful application in this direction. The vocabulary found in source code is typically rather technical and uses too many broken metaphors, such as "storing persons in a tree," that ontologies would be able to infer a meaning domain model without human supervision.

The use of information retrieval techniques for software comprehension dates back to the late eighties. Frakes and Nejmeh proposed to apply them to source code as if it were a natural language text corpus [68]. They applied an IR system based on keyword matching, which supported simple searches using wildcards and boolean operators. More recently, Antoniol *et al.* have published a series of papers on recovering code to documentation traceability [6, 7].

Maletic and Marcus were the first to propose using LSI (latent semantic indexing) to analyze software systems. In a first work they categorized the source files of the Mosaic web browser and presented in several follow ups other applications of LSI in software analysis [134]. Their work is a precursor of our work presented in Chapter 4, as they proved that LSI is a usable technique to compare software source documents. In follow up work, Marcus and Maletic used LSI to detect high-level conceptual clones, that is they go beyond just string based clone detection using the LSI capability to spot similar terms [136]. They select a known implementation of an abstract datatype, and manually investigate all similar source documents to find high-level concept clones. The same authors also used LSI to recover links between external documentation and source code by querying the source code with queries from documentation [135].

Kawaguchi *et al.* used LSI to categorize software systems into open-source software repositories [102]. Their approach uses the same techniques as ours, but with a different set up and other objectives. They present a tool that categorizes software projects in a source repository farm, that is they use entire software systems as the documents of their LSI space. They use clustering to provide overlapping categorizations of software, whereas we use clustering to partition the software into distinct topics. They use a visualization of they results with the objective to navigate among categorizations and projects, similar to the Softwarenaut tool [132], whereas we use visualizations to present an overview, including all documents and the complete partition, at one glance.

Marcus *et al.* employed LSI to detect concepts in source code [138]. They used LSI as a search engine and searched in the code the concepts formulated as queries. Their article also gives a good overview of the related work. Marcus *et al.* also use LSI to compute the cohesion of a class based on the semantic similarity of its methods [137]. In our work, we extend this approach and illustrate on the correlation matrix both semantic similarity within a cluster and the semantic similarity between clusters.

De Lucia *et al.* introduce strategies to improve LSI-based traceability detection [47]. They use three techniques of link classification: taking the top-n search results, using a fixed or a variable threshold. Furthermore they create separate LSI spaces for different document categories and observe better results that way, with best results on pure natural language spaces. Lormans and

Deursen present two additional links classification strategies [130], and discuss open research questions in traceability link recovery.

Di Lucca *et al.* also focus on external documentation, automatically assigning maintenance requests to teams [56]. They compare approaches based on pattern matching and clustering to information retrieval techniques, of which clustering performs better.

Huffman-Hayes *et al.* compare the results of several information retrieval techniques in recovering links between document and source code to the results of a senior engineer [93]. The results suggest that automatic recovery performs better than human analysis, both in terms of precision and recall and with comparable signal-to-noise ratio.

Čubranić *et al.* build a searchable database with artifacts related to a software system, both source code and external documentation [46]. They use a structured meta model, which relates bug reports, news messages, external documentation and source files to each other. Their goal is to support software engineers, especially those new to a project, with a searchable database of what they call "group memory". They implemented their approach in an eclipse plug-in called Hipikat.

Anslow *et al.* [5] visualized the evolution of words in class names in Java version 1.1 and Java version 6.0. They illustrated the history in a combined word cloud that contains terms from both versions. Each word is printed twice, font size represents word frequency and color the corpus. As such they compared word counts, which assumes normal distribution and is thus not as sound as using log-likelihood ratios (see Chapter 5).

Linstead *et al.* [127] analysed the vocabulary of over 10,000 open source projects from Sourceforge and Apache. They provide strong evidence of power-law behavior for word distribution across program entities. In addition, they analyse the vocabulary of structural entities (class, interface, method, field) and report the top-10 most frequent terms, as well as the top-10 unique terms for each structural category.

Lexical information of source code has further proven useful for various tasks in software engineering (for example [9, 137, 159]). Many of these approaches apply Latent Semantic Indexing and inverse-document frequency weighting, which are well-accpeted techniques in Information Retrieval but are according to Dunning "only justified on very sketchy grounds [64]."

Baldi *et al.* [18] present a theory of aspects (the programming language feature) as latent topics. They apply Latent Dirichlet Analysis (LDA) to detect topic distributions that are possible candidates for aspect-oriented programming. They present the retrieved topics as a list of the 5 most likely words.

2.3 Social Information

In this section we first discuss work related to trustability in code search engines and the work related to modelling developer expertise and bug triage.

Since the rise of internet-scale code search engines, searching for reusable

source code has quickly become a fundamental activity for developers [15]. However, in order to establish search-driven software reuse as a best practice, the cost and time of deciding whether to integrate a search result must be minimized. The decision whether to reuse a search result or not should be quickly taken without the need for careful (and thus time-consuming) examination of the search results.

Credibility (sometimes also referred to as trustability) is a major concern for reusing code. When a developer reuses code from an external sources he has to trust the work of external developers who are unknown to him. This is not to be confused with trustworthy computing, where clients are concerned with security and reliability of a computation service.

For a result to actually be helpful and serve the purpose originally pursued with the search it is not enough to just match the entered keywords. It is essential that the developer know at least the license under which certain source code was published, otherwise he will not be able to use it legally. Furthermore, it is very helpful to know from which project a search result is taken when assessing its quality. User studies have shown that developers rely on both technical and human clues to assess the trustability of search results [73]. For example developers will prefer results from well-known open source projects over results from less popular projects.

The issue of providing meta-information alongside search results and thereby increasing trustabilty has not been widely studied and we are trying to address this with our work.

In recent years special search engines for source code have appeared, namely Google Code Search [9], Krugle [10] and Koders [11]. They all focus on full-text search over a huge code base, but lack detailed information about the project. Search results typically provide a path to the version control repository and little meta-information on the actual open source project; often, even such basic information as the name and homepage of the project are missing.

Sourcerer [12] by Bajracharya *et al.* [16] and Merobase [13] by Hummel *et al.* [94] are research projects with an internet-scale code search-engine. Both provide the developer with license information and project name. Merobase also provides a set of metrics such as cyclomatic- and Halstead complexity.

In addition to the web user interface, both Sourcerer and Merobase are also accessible through Eclipse plug-ins that allow the developer to write unit tests. These are then used as a special form of query to search for matching classes/ methods, that is classes that pass the tests[94]. Using unit tests as a form of formulating queries is a way of increasing technical trustability: Unit-tested search results are of course more trustable, however at the cost of a more time consuming query formulation (that is additionally writing the unit tests). The kind of results returned are also limited to clearly-defined and testable features.

[9] http://www.google.com/codesearch
[10] http://www.krugle.org
[11] http://www.koders.com
[12] http://sourcerer.ics.uci.edu
[13] http://www.merobase.org

A combination of technical trustability factors (for example unit tests) and human trustability factors might be promising future work.

Ichii *et al.* used collaborative filtering to recommend relevant components to users [96]. Their system uses browsing history to recommend components to the user. The aim was to help users make cost-benefit decisions about whether or not those components are worth integrating. Our contribution beyond the state-of-the-art is our focus on human factors and the role of cross-project contributors.

Mockus and Herbsleb [144] compute the experience of a developer as a function of the number of changes he has made to a software system so far. Additionally, they compute recent experience by weighting recent changes more than older ones. The experience is then used to model the expertise of a developer. Furthermore, they examine the time that a developer needs to find additional people to work on a given modification request. Based on the results, they report that finding experts is a difficult task.

Fritz *et al.* [71] report on an empirical study that investigates whether a programmer's activity indicates knowledge of code. They found that the frequency and recency of interaction indicates the parts of the code for which the developer is an expert. They also report on a number of indicators that may improve the expertise model, such as authorship, role of elements, and the task being performed. In our work, we use the vocabulary of frequently and recently changed code to build an expertise model of developers. By using the vocabulary of software changes, lexical information about the role of elements and the kind of tasks are included in our expertise model.

Siy *et al.* [171] present a way to summarize developer work history in terms of the files they have modified over time by segmenting the CVS change data of individual Eclipse developers. They show that the files modified by developers tend to change significantly over time. However, they found that most of the developers tend to work on files within the same directory. Gousios *et al.* [77] present an approach for evaluating developer contributions to the software development process based on data acquired from software repositories and collaboration infrastructures. However, their expertise model does not include the vocabulary of software changes and is thus not queryable using the content of bug reports. Alonso *et al.* [2] describe an approach using classification of the file paths of contributed source code files to derive the expertise of developers.

Schuler and Zimmerman [165] introduce the concept of usage expertise, which manifests itself whenever developers are using functionality, for example by calling API methods. They present preliminary results for the Eclipse project indicating that usage expertise is a promising complement to implementation expertise (such as our expertise model).

Mailing lists are a valuable source of developer expertise. It has been shown that the frequency with which software entities (functions, methods, classes, etc) are mentioned in the mail correlated with the number of times these entities are included in changes to the software [152]. It has been shown that about 70% of the vocabulary used in source code changes is found in mailing lists as well [22].

While research in the mining of software repositories has frequently ignored commits that include a large number of files; Hindle *et al.* [87] perform a case

study that includes the manual classification of large commits. They show that large commits tend to be perfective while small commits are more likely to be corrective. Commits are not normalized in our expertise model, thus the size of a commit may affect our model.

Hill *et al.* [86] present an automatic mining technique to expand abbreviations in source code. Automatically generated abbreviation expansions can be used to enhance software maintenance tools that utilize natural language information, such as our approach. If the same abbreviations are used in both souce code and bug reports then our approach is not affected by this issue, nevertheless we plan to include automatic abbreviation expansion as future work.

Currently our expertise model is limited to the granularity of commited software changes, a more fine grained acquisition of the model could be achieved by using a change-aware development environment [150, 163] that records developer vocabulary as the software is written.

Bettenburg *et al.* [26] present an approach to split bug reports into natural text parts and structured parts, that is source code fragments or stack traces. Our approach treats both the same, since counting word frequencies is applicable for natural-language text as well as source code in the same way.

Recommending experts to assign developers to bug reports is a common application of developer expertise models. Anvik *et al.* [11] build developers' expertise from previous bug reports and try to assign current reports based on this expertise. They label the reports. If a report cannot be labeled, it is not considered for training. Additionally, reports involving developers with a too low bug fixing frequency or involving developers not working on the project anymore are filtered. They then assign bug reports from a period of lower than half a year. To find possible experts for their recall calculation, they look for the developers who fixed the bug in the source code (by looking for the corresponding bug ID in the change comments). They reach precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively. However, they only achieve around 6% precision on the Gnu C Compiler project. The highest recall they achieve is on average 10% (Eclipse), 3% (Firefox) and 8% (gcc). Please note that the recall results are not directly comparable to ours, since they use different configurations of bug-related persons to compute recall.

Cubranic and Murphy [187] propose to use machine learning techniques to assist in bug triage. Their prototype uses supervised Bayesian learning to train a classifier with the textual content of resolved bug reports. This is then used to classify newly incoming bug reports. They can correctly predict 30% of the report assignments, considering Eclipse as a case study.

Canfora and Cerulo [41] propose an Information Retrieval technique to assign developers to bug reports and to predict the files impacted by the bug's fix. They use the lexical content of bug reports to index source files as well as developers. They do not use vocabulary found in source files, rather they assign to source files the vocabulary of related bug reports. The same is done for developers. For the assignments of developers, they achieve 30%–50% top-1 recall for KDE and 10% to 20% top-1 recall for the Mozilla case study.

Similar to our work, Di Lucca *et al.* use Information Retrieval approaches to

classify maintenance requests [56]. They train a classifier on previously assigned bug reports, which is then used to classify incoming bug reports. They evaluate different classifiers, one of them being a term-documents matrix using cosine similarity. However, this matrix is used to model the vocabulary of previous bug reports and not the vocabulary of developers.

Anvik and Murphy evaluate approaches that mine implementation expertise from a software repository or from bug reports [12]. Both approaches are used to recommend experts for a bug report. For the approach that gathers information from the software repository, a linkage between similar reports and source code elements is required. For the approach that mines the reports itself, amongst others, the commenters of the reports (if they are developers) are estimated as possible experts. Both approaches disregard inactive developers. Both recommendation sets are then compared to human generated expert sets for the bug report.

Minto and Murphy's Emergent Expertise Locator (EEL) [143] recommends a ranked list of experts for a set of files of interest. The expertise is calculated based on how many times which files have been changed together and how many times which author has changed what file. They validate their approach by comparing recommended experts for files changed for a bug fix with the developers commenting on the bug report, assuming that they "either have expertise in this area or gain expertise through the discussion" [143].

2.4 Story-telling Visualization

Story-telling visualizations are a branch of information visualization that has been popularized by the political information graphics of newspapers such as the New York Times and the Guardian. A story-telling visualization is supposed to invite its reader to get engaged with the visualized data by establishing a personal connection between the reader and the presented data [167].

Analyzing the way developers interact with the system has only attracted few research. A visualization similar to the *Ownership Map* is used to visualize how authors change a wiki page by Viega and Wattenberg [188].

Xiaomin Wu *et al.* visualize [195] the change log information to provide an overview of the active places in the system as well as of the authors activity. They display measurements like the number of times an author changed a file, or the date of the last commitment.

Measurements and visualization have long been used to analyze how software systems evolve. Ball and Eick [19] developed multiple visualizations for showing changes that appear in the source code. For example, they show what is the percentage of bug fixes and feature addition in files, or which lines were changed recently. Eick *et al.* proposed multiple visualizations to show changes using colors and a third dimension [65]. Chuah and Eick proposed a three visualizations for comparing and correlating different evolution information like the number of lines added, the errors recorded between versions, number of people working etc. [44].

Rysselberghe and Demeyer use a scatter plot visualization of the changes to provide an overview of the evolution of systems and to detect patterns of change[185]. Jingwei Wu *et al.* use the spectrograph metaphor to visualize how changes occur in software systems [194]. They used colors to denote the age of changes on different parts of the systems. Jazayeri analyzes the stability of the architecture [98] by using colors to depict the changes. From the visualization he concluded that old parts tend to stabilize over time.

Lanza and Ducasse visualize the evolution of classes in the Evolution Matrix [119]. Each class version is represented using a rectangle. The size of the rectangle is given by different measurements applied on the class version. From the visualization different evolution patterns can be detected such as continuous growth, growing and shrinking phases etc.

Another relevant reverse engineering domain is the analysis of the co-change history. Gall *et al.* aimed to detect logical coupling between parts of the system [72] by identifying the parts of the system which change together. They used this information to define a coupling measurement based on the fact that the more times two modules were changed at the same time, the more they were coupled. Zimmerman *et al.* aimed to provide mechanism to warn developers about the correlation of changes between functions. The authors placed their analysis at the level of entities in the meta-model (for example methods) [198]. The same authors defined a measurement of coupling based on co-changes [197]. Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graph [83].

2.5 Spatial Representation

In this section we discuss work related to the spatial representation of abstract information spaces such as source code. Using multidimensional scaling to visualize information based on the metaphor of cartographic maps is by no means a novel idea. *Topic maps*, as they are called, have a longstanding tradition in information visualization [190]. The work in this paper was originally inspired by Michael Hermann's and Heiri Leuthold's work on the political landscapes of Switzerland [85]. Reader who are not know knowledgeable in German may refer to Hermann's recent TED talk on his work, which is available online[14].

In the same way, stable layouts have a long history in information visualization, as a starting point see for example the recent work by Frishman and Tal on online dynamic graph drawing [69]. They present an online graph drawing approach, which is similar to the online pipeline presented in this work.

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 [193]. The visualization is part of a larger toolset that uses a variety of algorithms to cluster terms in

[14] http://tedxzurich.com/2010/09/05/michael-hermann-visualizes-politics

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.

In the software visualization literature however, topic maps are rarely used. Except for the use of graph splatting in RE Toolkit by Telea *et al.* [177], we are unaware of their prior application 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.

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

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 (for example 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 notice the change, 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) [183]. 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 [3]. They report that the layout scheme that groups architecturally related classes together yields best results. Which is consistent with our user study on software maps presented in Chapter 7. 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 [40]. 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 [190, 101]. Noack *et al.* offer a good starting point for applying graph drawing to software visualization [149]. Jucknath-John *et al.* present a technique to achieve stable graph layouts over the evolution of the displayed software system [99], thus achieving consistent layout, while sidestepping the issue of reflecting meaningful position or distance metrics.

Graph splatting is a variation of graph drawing, which produced visualizations that are very similar to thematic maps [184]. 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 [177]. However, they are not concerned with stable layouts. Each run of their tool may yield a different layout.

Treemaps represent tree-structured information using nested rectangles [190]. 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 [20]. Their approach creates aesthetically more appealing treemaps, reducing the number of narrow tessels. There are some treemap variations (for example 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 [120, 121]. 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 then 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 for example [78]). 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 desiderate for a stable and consistent layout.

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

MetricView is an exploratory environment featuring UML diagram visualizations [180]. The third dimension is used to extend UML with polymetric views [120]. The diagrams use arbitrary layout, so do not reflect meaningful distance or position.

White Coats is an explorative environment also based on the notion of polymetric views [142]. The visualizations are three-dimensional with position and visual-distance of entities given by selected metrics. However they do not incorporate the notion of a consistent layout.

CGA Call Graph Analyser is an explorative environment that visualizes a combination of function call graph and nested modules structure [31]. The tool employs a $2\frac{1}{2}$-dimensional approach. To our best knowledge, their visualizations

use an arbitrary layout.

CodeCity is an explorative environment building on the city metaphor [191]. CodeCity employs the nesting level of packages for their city's elevation model, and uses a modified tree layout to position the entities, that is packages and classes. Within a package, elements are ordered by size of the element's visual representation. Hence, when changing the metrics mapped on width and height, the overall layout of the city changes, and thus, the consistent layout breaks.

VERSO is an explorative environment that is also based on the city metaphor [118]. Similar to CodeCity, VERSO employs a treemap layout to position their elements. Within a package, elements are either ordered by their color or by first appearance in the system's history. As the leaf elements have all the same base size, changing this setting does not change the overall layout. Hence, they provide consistent layout, however within the spatial limitations of the classical treemap layout.

Software Cities by Steinbrückner *et al.* is a more recent explorative environment built on the city metaphor [174]. Software cities use a custom layout mechanism that is consistent and stable over time. New packages are added as orthogonal streets to the city layout, new classes are added to existing streets by extending them in length. While not being stable in terms of mathematical metrics, the resulting layout is perceived as being stable over time by humans.

Chapter 3

User Study on API Learning

Modern software development requires a large investment in learning application programming interfaces (APIs). Recent research found that the learning materials themselves are often inadequate: developers struggle to find answers beyond simple usage scenarios. Solving these problems requires a large investment in tool and search engine development. To understand where further investment would be the most useful, we ran a study with 19 professional developers to understand what a solution might look like, free of technical constraints. In this chapter, we report on design implications of tools for API learning, grounded in the reality of the professional developers themselves. The reoccurring themes in the participants' feedback were trustworthiness, confidentiality, information overload and the need for code examples as first-class documentation artifacts.

Modern application programming interfaces (APIs) allow developers to reuse existing components. API learning is a continuous process. Even when a developer makes a large initial investment in learning the API, for example, by reading books or going through online tutorials, the developer will continue to consume online material about the API throughout the development process. These materials include reference documentation from the API provider, sample code, blog posts, and forum questions and answers.

Indeed, seeking online API information has become such a pervasive part of modern programming that emerging research tools blend the experiences of the browser and the development environment. For example, Codetrail automatically links source code and the web pages viewed while writing the code [76]. Blueprint allows a developer to launch a web query from the development environment and incorporate code examples from the resulting web pages [36]. While these new tools help reduce the cost of (re)finding relevant pages and incorporating information from them, this addresses only a portion of developers' frustrations. In a recent study of API learning obstacles among professional developers, Robillard found that the learning materials themselves are often in-

adequate [164]. Bajracharya and Lopes analysed a year's worth of search queries and found that current code search engines address only a subset of developers needs [14]. For example, developers struggled to find code examples beyond simple usage scenarios, to understand which parts of an API support which programming tasks, and to infer the intent behind the API's design. Solving these systematic problems requires a large investment, either in the API provider's official documentation, the API users' community-based documentation, or in the search engines that unite the two [36, 91]. Any of these changes is difficult and expensive.

To understand where further investment would be the most useful, we ran a study with 19 professional developers from Microsoft Corporation, with the goal of understanding what an "ideal" solution might look like, free from technical constraints. We invited randomly chosen members of a corporate email list of Silverlight users to participate in one-hour sessions for small gratuities. Silverlight is a large API for creating web applications, with hundreds of classes for data persistance, data presentation, and multimedia. All participants were male with an average of 12.2 years of professional experience.

Borrowing from participatory design, we asked the participants to act as our partners in designing a new user experience for learning Silverlight. We ran two types of sessions. In the first, we interviewed participants to learn their common learning materials and most challenging learning tasks and then asked them to sketch a design for a new learning portal. We compiled these ideas into five exploratory designs. In the second type of session, we ran focus groups to get feedback on our descriptions of their learning tasks and the five designs.

This chapter's main contributions are a compilation of design implications for API learning tools, grounded in the reality of the professional developers themselves. We report on the recurring themes in the participants' feedback: trustworthiness, confidentiality, information overload and the need for code examples as first-class documentation artifacts.

3.1 First Study: Current Practice

In the first type of study session, we met singly with nine participants, and ran each through three activities. First, we asked the participant to describe all the materials he used for learning Silverlight, as we recorded them on the whiteboard. Next, we asked him to consider this as a set of "ingredients" and to sketch a design for a learning portal that presents some or all of these ingredients to help developers learn Silverlight. Finally, we asked him to review the design by comparing the experience of learning Silverlight by using the design versus his own experience learning Silverlight.

3.1.1 Learning Sources

We asked the participant to describe all the materials he used for learning Silverlight, as we recorded them on the whiteboard. Some of the learning sources

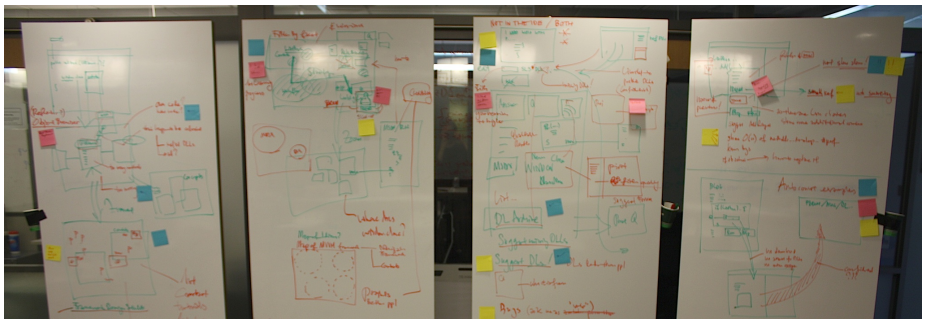


Figure 3.1: The five solution designs as narrated to the participants (from left to right): Zoomable UML, Concept Map, Faceted Search, and on the last panel Rich Intellisense (above) and Interactive Snippets (below). Pen color has been used to distinguish the designs (green) from the participant's input (red). The stick notes are the participant's votes.

are obvious and readily reported by participants, such as books and web search. To learn about non-obvious learning sources, we asked developers "did you ever find an answer to a technological question that is not listed here," which led to answers like reverse engineering or social networking. Their reported learning sources are the following:

"Off the top of my head" is by far the most common way developers find answers on the job. Most participants reported that they set aside dedicated time for learning. Typical off-the-job learning sources are: lurking on mailing lists and forums, watching videos and reading books. Most knowledge however is based on experience and acquired through learning-by-doing on the job. One participant refers to this a *"growing your own folklore."*

Web search was reported by all participants as the first place to go when they have an information need. Among the search results participants are typically looking for are: blog posts, discussion forums, official reference documentation, mailing list archives, bug reports and source repositories (listed in order of typical access patterns). Participants prefer results with code examples over results without code examples, which is supported by existing research on API learning barriers [164].

Intellisense (that is auto-completion of identifiers) was reported as a tool for the discovery of unknown APIs by all developers. One participant called this *"digging through the namespaces."* Discovering unknown APIs is an appropriation of auto-completion, originally conceived to help recall names from familiar APIs.

Prototyping, reverse engineering and many more forms of tinkering were reported by all participants as a last resort when all above sources failed to

provide an answer. Some participants even resort to reverse engineering when documentation is available, as they prefer source code over natural language documentation. Developers typically use prototyping both as an explorative tool and to verify hypotheses about the APIs. All participants reported that having to *"get your hands dirty"* is an integral part of their learning experience.

Asking another person was reported by most participants as a last resort. Developers follow a "due diligence" process before asking another person for an answer. It is important to them to have put enough personal effort into finding an answer before asking on a mailing list or reaching out to a person from their social network. Also, they reported to prefer immediate results, such as those provided by web search, over waiting for the reply of asynchronous communication such as email and discussion forums.

These findings are consistent with Robillard's study of learning obstacles [164], but provide a more complete catalog of learning materials. Both studies found that developers strive to stay within the programming patterns and use cases that the API provider intends (even when that intent is undocumented) and that developers typically lack documentation when using an API for a less common task. Somewhat surprisingly, we found that developers prefer the community-based learning materials on the web, like blogs, forum posts, and tutorials, over more "authoritative" learning material, like books and reference documentation. Developers also prefer active but potentially time-consuming information seeking, like iterative web search and reverse engineering, to waiting for answers from others, because they perceive the answers as more immediate.

3.1.2 Learning Categories

Based on the design sketches that participants produced, we elicited three broad categories of learning tasks:

Technology selection is about learning about an API's fundamental capabilities (*"Can Silverlight play video in this codec?"*) and about comparing capabilities (*"Is DirectX or Silverlight better for my game?"*). Sometimes the selection decision is about growing skills rather than project requirements.

Mapping task to code includes both discovery of unfamiliar APIs as well as remembering relevant identifier names in previously learned APIs. Getting an answer to this type of questions typically falls in two phases. Initially developers search based on natural language task descriptions (e.g. *"how to implement a zoomable canvas"*) and skim through many search results to stumble on relevant identifier names. Once they have a concrete identifier, their search behavior becomes more focused and may be as simple as looking up the identifier's reference documentation.

Going from code to better code is a major concern of professional developers. All participants reported that they spend considerable effort getting answers to performance questions. Other use cases are robustness and idiomatic, that is intended, use of an API, in particular with regard to breaking changes of newly released API versions or different target platforms.

The kind of learning categories impacts the preferred learning sources and strategies of developers. For technology selection, participants sometimes use web search to learn about available technologies, but eventually prefer personal recommendations from their social network. For mapping task to code, participants strongly prefer search results with code examples over those without code examples. For getting to better code however, such as troubleshooting a performance problem, participants prefer solving the problem themselves (including reverse engineering) but sometimes ask others to double-check the answers they find.

3.2 Second Study: Solution Elements

For the second type of session, we compiled the user feedback from the first sessions into five exploratory designs. We ran 10 participants in three focus groups (with three, three, and four members) and asked them to provide feedback on the five designs. In each session, we drew each design on its own whiteboard and encouraged participants to ask questions, provide feedback, and to add their own ideas as we explained the design.

Figure 3.1 shows a photograph of the whiteboards with the five designs, taken at the end of a focus group's session. In the following the designs are described in the order they were presented to the participants in that session:

Design: Zoomable UML This design draws from the spatial software representation of CodeCanvas [54] and addresses answering complex reachability questions [122] as you code. The design extends the IDE with a zoomable UML diagram. The diagram opens zoomed on locally reachable types of the API and shows their dependencies and interaction. The user can zoom out to get a larger picture of the API, up to the level of namespaces.

Design: Concept Map The API is presented as a zoomable map, organized around programming domain concepts (e.g. "controls", "media content"). As the user zooms in, the concepts become more refined (e.g. "streaming video"). At the lowest zoom level, the map shows web-based content about that concept, including blogs, forum posts, tutorials, and the people who author these. The map is searchable and keeps track of user interaction as well as the user's learning progress. Users can bookmark locations and share their bookmarks. Documentation editors can use the same feature to share tutorials as "sightseeing tours."

Design: Faceted Search This design unifies web search and asking people questions. The user types a question into a textbox. As she types, related search results are pulled in from various sources (web sites, bug reports, code examples, mailing list archives, etc). Search results are grouped by facets, such as type of sources, type of content or semantic concepts. Besides the results, a tag cloud appears with extracted identifier names. Search results are summarized using code examples, if possible. In addition, the results include suggested people and mailing lists that are experts on the topic of the questions, to which the question can be posted.

Design: Rich Intellisense This design extends auto-completion of identifiers with search results that are automatically pulled from the world wide web. The results are "localized" to the current context of the IDE, such as imported libraries and reachable types [92]. Results are shown in the same pop-up windows as the auto-completion suggestions. If possible, search results are displayed as code examples ready for incorporating into the code, as in Brandt et al [36].

Design: Interactive Snippets This design attaches an execution context to code examples on the web. Code examples include hidden meta-information with all context that is required to execute. Examples are editable, debuggable and can be executed live in the browser. With a single click, users can download examples into their IDEs. Similarly, users can upload the code in their IDE as runnable examples on the web, for inclusion in blogs or discussion forums.

3.3 Feedback

After we explained all five designs, we then handed each participant a pen and sticky notes and gave them 10 minutes to annotate the designs, either with a blank sticky note to mean "I like this part" or with their own comments (typical ones were smiley faces, frowny faces, "NO", etc).

The votes are summarized in Table 9.1: the most popular designs are "Faceted Search" and for learning activities the "Concept Map" design. Participants downvoted the "Zoomable UML" and "Rich Intellisense" due to concerns about information overload, the same happening with "Interactive Snippets" due to concerns about missing confidentiality.

There were several recurring themes in our participants' feedback which cut across the various designs. The four top-most recurring themes are discussed and summarized as design implications for tool builders in the following:

3.3.1 Code Examples

We got very positive feedback on the emphasis on code examples and identifier names in the "Faceted Search" design. Participants prefer results with code examples over results without code examples, which is supported by existing research on API learning barriers [164]. When mapping a task to code, developers

Design	Up-Votes	Down-Votes
Zoomable UML	⋆⋆⋆⋆	∗∗∗∗∗
Concept Map	⋆⋆⋆⋆⋆⋆	∗∗∗
Faceted Search	⋆⋆⋆⋆⋆⋆⋆⋆⋆	∗
Rich Intellisense	⋆⋆⋆	∗∗∗∗∗∗∗
Interactive Snippets	⋆⋆⋆	∗

Table 3.1: At the end of the second type of sessions, participants voted with sticky notes for the designs. Faceted Search received the most up votes, Rich Intellisense the most down votes.

typically use web search and linearly go through all results until they find one with a code example or an identifier; often repeating this process a dozen times until they find a working answer. Participants liked about the facetted search design that it extracts code examples and identifiers from top search results. One participant even said that the summary tag cloud with identifiers, by itself, would be reason to use it.

Implication for tool builders: Developers need the heterogeneous learning materials that web search provides, but want it to be more targeted and organized. Search engines for API learning should extract code examples and identifiers found in natural text documents, and present them to the developers in a more accessible way. This implication is supported by related work on code examples [36, 91, 92].

3.3.2 Credibility

Credibility of web sources appeared as a major concern with all designs that included content taken from the web. For the participants, credibility is mostly a function of where the information comes from. For example, participants reported that search results from blogs are often more relevant, but typically less credible than official reference documentation. They also rely on the social reputation of its source rather than technical factors, which supports existing research [82]. In particular with the "Faceted Search" design, which automatically summarizes search results, participants emphasized the importance of seeing the information source to judge credibility.

Implication for tool builders: Tools should show both credibility and relevance when presenting search results, so that the developers can make an informed decision when using API information and code examples from the web. To assess the credibility of API information tools should prefer social factors, such as the credibility of the information's author, rather than technical statistics, such as code metrics.

3.3.3 Confidentiality

Confidentiality appeared as a major concern with all designs that share local information with a global audience. In particular with the "Interactive Snippets" design, which publishes an example's execution context on the web, participants were concerned with leaking proprietary information, like the use of certain libraries. One participant was also concerned that publicly inquiring about technologies could accidentally reveal business strategies.

Implication for tool builders: When automatically sharing local information with the web, tools must be careful to protect proprietary information, such as confidential code and libraries being used. Tools should give developers full control over shared information, for example by letting them review the list of automatically included terms before issuing the search query. Or alternatively, only sharing information that is on a user controlled white list.

3.3.4 Information Overload

Information overload was the major reason why participants rejected the "Zoomable UML" and the "Rich Intellisense" designs. We got strong feedback that pulling more information into the IDE is not welcome unless it is highly task- and context specific. Participants were also concerned that adding more features to Intellisense's popup will use too much screen real estate and slow down the IDE.

Implication for tool builders: Any tool that pulls additional information into the IDE must be highly selective and should only show information that is specific to the developer's current task and context. The ability to further filter down the information is crucial, as well as not slowing down the IDE and using screen real estate sparingly.

3.3.5 Threats to validity

We selected all participants from the same corporation, whose common hiring practices and corporate culture may bias the results. In particular, the participants all work for the same company that produces the Silverlight API, which gives the participants unique access to the API creators. Therefore, the participants may not be representative of all professional developers. Nonetheless, participants mostly accessed public learning outside the company and many expressed hesitation about asking questions of fellow employees for fear of harming their reputation. The study is also based on a single API. While this choice allowed us to compare participants' experiences and gave them common ground during the focus groups, there may be issues in learning Silverlight that do not generalize to other APIs.

3.4 Conclusion

Web search is the predominant form of information seeking, but in many cases is frustrating and error-prone. Developers need the heterogeneous learning materials that web search provides, but want it to be more targeted and organized. Therefore, API learning tools that bring web search and development environments closer together 1) should leverage examples and identifiers found in natural text documents as first-class results, 2) should communicate the credibility of aggregated results, 3) most not shared confidential information without the user's consent, and 4) should filter search results by task and context to avoid information overload.

Chapter 4

Lexical Clustering

Aim	Refinding and discovering topics.
Reach	Local codebase of a system.
Clues	Lexcial (established through lexical information).
Query	Visual analytics and fuzzy keyword search.

Keyword matching and regular expressions are powerful means for code orientation by lexical clues. However, current tool support fails to meet the developer needs when following up on fuzzy lexical clues. For example, for refinding tasks it may be that developers do not recall the exact name, or even more common for discovery tasks developers can typically only guess which name other developers have picked for the concept that they are looking for. Just the same, when attempting to find all implementations of the same concept, often the source code uses synonymous but not identical identifier names. And further, when encountering a system for the first time, developers have a need to cluster the system by topics so they can start establishing a mental model of the services provided by the system and of how these services depend upon one another.

In this chapter we present an approach to model a system's lexical information in a fuzzy text model that resolves synonymy and polysemy with unsupervised learning. No user input or ontologies are required to resolve ambiguous lexical clues. Software engineers make frequent use of broken metaphors (as for example "storing persons in a tree") so that common natural language ontologies fall short of being applicable to lexical information found in source code. We use the fuzzy text model to cluster the parts of a system by topic, and visualize the topics using correlation matrices and *distribution maps* to illustrate the distribution of topics of the static structure of the system. Furthermore, even though not discussed in this chapter, our approach allows to query the system with fuzzy search terms that are able resolve synonymy and polysemy.

Acquiring knowledge about a software system is one of the main activities in software reengineering. It is estimated that up to 60 percent of software

maintenance is spent on comprehension [1]. This is because a lot of knowledge about the software system and its associated business domain is not captured in an explicit form. Most approaches that have been developed focus on program structure [60] or on external documentation [133, 7]. However, there is another fundamental source of information: the developer knowledge contained in identifier names and source code comments.

> *The informal linguistic information that the software engineer deals with is not simply supplemental information that can be ignored because automated tools do not use it. Rather, this information is fundamental. [...] If we are to use this informal information in design recovery tools, we must propose a form for it, suggest how that form relates to the formal information captured in program source code or in formal specifications, and propose a set of operations on these structures that implements the design recovery process* [28].

Languages are a means of communication, and programming languages are no different. Source code contains two levels of communication: human-machine communication through program instructions, and human to human communications through names of identifiers and comments.

Many of the existing approaches in Software Comprehension focus on program program structure or external documentation. However, by analyzing formal information the informal semantics contained in the lexical information of source code are overlooked. To understand software as a whole, we need to enrich software analysis with the developer knowledge hidden in the code naming. This chapter proposes the use of information retrieval to exploit linguistic information found in source code, such as identifier names and comments. We introduce *Lexical Clustering*, a technique based on Latent Semantic Indexing and clustering to group source artifacts that use similar vocabulary. We call these groups *lexical clusters* and we interpret them as *linguistic topics* that reveal the intention of the code. We provide automatically retrieved labels, and use a visualization to illustrate how they are distributed over the system. Our approach is language independent as it works at the level of identifier names. To validate our approach we applied it to several case studies, two of which we present in this chapter.

We call our clusters *linguistic topics* since they are derived from language use. Some linguistic topics do map to the domain and others do map to application concepts, however, this mapping is never complete. We refrain from speaking of "linguistic concepts" since there is no guarantee that lexical clustering locates all or even any externally defined domain concept.

In this chapter, we use information retrieval techniques to *derive topics from the lexical information at the source code level*. Apart from external documentation, the location and use of source-code identifiers is the most frequently consulted source of information in software maintenance [106]. The objective of our work is to analyze software without taking into account any external documentation. In particular we aim at:

- **Providing a first impression of an unfamiliar software system**.

A common pattern when encountering an unknown or not well known software for the first time is "Read all the Code in One Hour" [55]. Our objective is to support this task, and to provide a map with a survey of the system's most important topics and their location.

- **Revealing the developer knowledge hidden in identifiers.** In practice, it is not external documentation, but identifer names and comments where developers put their knowledge about a system. Thus, our objective is not to locate externally defined domain concepts, but rather to derive topics from the actual use of lexical information in source code.

- **Enriching Software Analysis with informal information.** When analyzing formal information (for example structure and behavior) we get only half of the picture: a crucial source of information is missing, namely, the vocabulary contained in the lexical information of source code. Our objective is to reveal components or aspects when, for example, planning a large-scale refactoring. Therefore, we analyze how the code naming compares to the code structure: What is the distribution of linguistic topics over a system's modularization? Are the topics well-encapsulated by the modules or do they cross-cut the structure?

Our approach is based on Latent Semantic Indexing (LSI), an information retrieval technique that locates linguistic topics in a set of documents [49, 138]. We apply LSI to compute the linguistic similarity between source artifacts (for example packages, classes or methods) and cluster them according to their similarity. This clustering partitions the system into linguistic topics that represent groups of documents using similar vocabulary. To identify how the clusters are related to each other, we use a correlation matrix [126]. We employ LSI again to automatically label the clusters with their most relevant terms. And finally, to complete the picture, we use a map visualization to analyze the distribution of the concepts over the system's structure.

We implemented this approach in a tool called Hapax[1], which is built on top of the Moose reengineering environment [59, 148], and we apply the tool to several case studies, two of which are presented in this work: JEdit[2] and JBoss[3].

This chapter is based on our previous work, in which we first proposed lexical clustering (back then still called "semantic clustering" [109]. The main contributions of the current chapter are:

- *Topic distribution analysis.* In our previous work we introduced lexical clustering to detect linguistic topics given by parts of the system that use similar vocabulary. We complement the approach with the analysis of how topics are distributed over the system using a Distribution Map [58].

[1] The name is derived from the term *hapax legomenon*, that refers to a word occurring only once a given body of text.

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

[3] http://www.JBoss.org/

- *Case studies.* In our previous work, we showed the results of the clustering and labeling on different levels of abstraction on three case studies. In this chapter we report on other two case studies.

4.1 Latent Semantic Indexing

As with most information retrieval techniques, Latent Semantic Indexing (LSI) is based on the vector space model approach. This approach models documents as bag-of-words and arranges them in a term-document matrix A, such that $a_{i,j}$ equals the number of times term t_i occurs in document d_j.

LSI has been developed to overcome problems with synonymy and polysemy that occurred in prior vectorial approaches, and thus improves the basic vector space model by replacing the original term-document matrix with an approximation. This is done using singular value decomposition (SVD), a principal components analysis (PCA) technique originally used in signal processing to reduce noise while preserving the original signal. Assuming that the original term-document matrix is noisy (the aforementioned synonymy and polysemy), the approximation is interpreted as a noise reduced – and thus better – model of the text corpus.

As an example, a typical search engine covers a text corpus with millions of web pages, containing some tens of thousands of terms, which is reduced to a vector space with 200-500 dimensions only. In Software Analysis, the number of documents is much smaller and we typically reduce the text corpus to 20-50 dimensions.

Even though search engines are the most common uses of LSI [23], there is a wide range of applications, such as automatic essay grading [67], automatic assignment of reviewers to submitted conference papers [63], cross-language search engines, thesauri, spell checkers and many more. In the field of software engineering LSI has been successfully applied to categorized source files [134] and open-source projects [102], detect high-level conceptual clones [136], recover links between external documentation and source code [47, 137] and to compute the class cohesion [137]. Furthermore LSI has proved useful in psychology to simulate language understanding of the human brain, including processes such as the language acquisition of children and other high-level comprehension phenomena [117].

Figure 4.1 schematically represents the LSI process. The document collection is modeled as a vector space. Each document is represented by the vector of its term occurrences, where terms are words appearing in the document. The term-document-matrix A is a sparse matrix and represents the document vectors on the rows. This matrix is of size $n \times m$, where m is the number of documents and n the total number of terms over all documents. Each entry $a_{i,j}$ is the frequency of term t_i in document d_j. A geometric interpretation of the term-document-matrix is as a set of document vectors occupying a vector space spanned by the terms. The similarity between documents is typically defined as the cosine or inner product between the corresponding vectors. Two documents

are considered similar if their corresponding vectors point in the same direction.

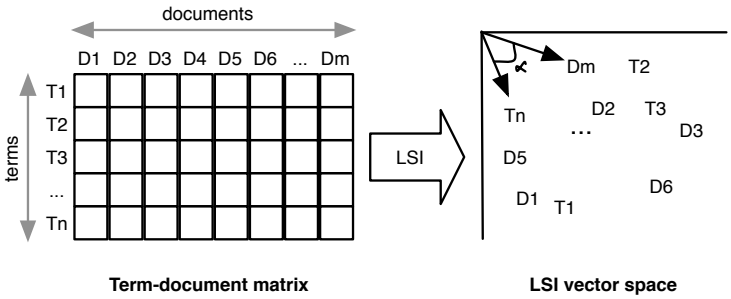


Figure 4.1: LSI takes as input a set of documents and the terms occurrences, and returns as output a vector space containing all the terms and all the documents. The similarity between two items (that is terms or documents) is given by the angle between their corresponding vectors.

LSI starts with a raw term-document-matrix, weighted by a weighting function to balance out very rare and very common terms. SVD is used to break down the vector space model into fewer dimensions. This algorithm preserves as much information as possible about the relative distances between the document vectors, while collapsing them into a much smaller set of dimensions.

SVD decomposes matrix A into its singular values and its singular vectors, and yields – when truncated at the k largest singular values – an approximation A' of A with rank k. Furthermore, not only the low-rank term-document matrix A' can be computed but also a term-term matrix and a document-document matrix. Thus, LSI allows us to compute term-document, term-term and document-document similarities.

As the rank is the number of linear-independent rows and columns of a matrix, the vector space spanned by A' is of dimension k only and much less complex than the initial space. When used for information retrieval, k is typically about 200-500, while n and m may go into millions. When used to analyze software on the other hand, k is typically about $20-50$ with vocabulary and documents in the range of thousands only. And since A' is the best approximation of A under the least-square-error criterion, the similarity between documents is preserved, while in the same time mapping lexically related terms on one axis of the reduced vector space and thus taking into account synonymy and polysemy. In other words, the initial term-document-matrix A is a table with term occurrences and by breaking it down to much fewer dimension the latent meaning *must* appear in A' since there is now much less space to encode the same information. Meaningless occurrence data is transformed into meaningful concept information.

4.2 Lexical Clustering: Grouping Source Documents

The result of applying LSI is a vector space, based on which we can compute the similarity between both documents or terms. We use this similarity measurement to identify topics in the source code.

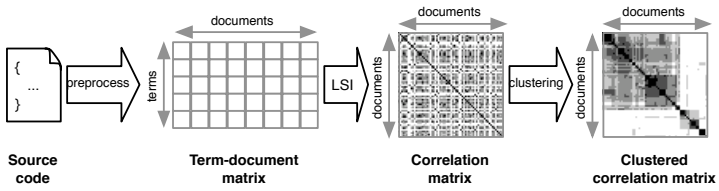


Figure 4.2: Lexical clustering of software source code (for example classes, methods).

Figure 4.2 illustrates the first three steps of the approach: preprocessing, applying LSI, and clustering. Furthermore we retrieve the most relevant terms for each cluster and visualize the clustering on a 2D-map, thus in short the approach is:

1. *Preprocessing the software system.* In subsection 4.2.1, we show how we break the system into documents and how we build a term-document-matrix that contains the lexical information of the system.

2. *Applying Latent Semantic Indexing.* In subsection 6.2.1 we use LSI to compute the similarities between source code documents and illustrate the result in a correlation matrix [126].

3. *Identifying topics.* In subsection 4.2.3 we cluster the documents based on their similarity, and we rearrange the correlation matrix. Each cluster is a lexical topic.

4. *Describing the topics with labels.* We use LSI again to retrieve for each cluster the top-n most relevant terms. For more information on labeling please refer to Chapter 5.

5. *Comparing the topics to the structure.* In Section 4.3 we illustrate the distribution of topics over the system on a Distribution Map [58].

4.2.1 Preprocessing the Software System

When we apply LSI to a software system we partition its source code into documents and we use the lexical information found therein as terms. The system can be split into documents at any level of granularity, such as packages

or classes and methods. Other slicing solutions are possible as well, for example execution traces [114], or we can even use entire projects as documents and analyze a complete source repository [102].

To build the term-document-matrix, we extract the lexical information from the source code: we use both identifier names and the content of comments. Natural language text in comments is broken into words, whereas compound identifier names are split into parts. As most modern naming conventions use camel case, splitting identifiers is straightforward: for example *FooBar* becomes *foo* and *bar*.

We exclude common stopwords from the vocabulary, as they do not help to discriminate documents. In addition, if the first comment of a class contains a copyright disclaimer, we exclude it as well. To reduce words to their morphological root we apply a stemming algorithm: for example *entity* and *entities* both become *entiti* [157]. And finally, the term-document matrix is weighted with *tf-idf* to balance out the influence of very rare and very common terms [62].

When preprocessing object-oriented software systems we take the inheritance relationship into account as well. For example, when applying our approach at the level of classes, each class inherits some of the vocabulary of its superclass. If a method is defined only in the superclass we add its vocabulary to the current class. Per level of inheritance a weighting factor of $w = 0.5$ applies to the term occurrences, to balance out between the abstractness of high level definitions and concrete implementations.

4.2.2 Using Latent Semantic Indexing to Build the Similarity Index

We use LSI to extract linguistic information from the source code, which results in an LSI-index with similarities between source documents (that is packages, classes or methods). Based on the index we can determine the similarity between source code documents. Documents are more similar if they cover the same topic, terms are more similar if they denote related topics.

In the vector space model there is a vector for each document. For example, if we use methods as documents, there is a vector for each method and the cosine between these vectors denotes the lexical similarity between the methods. In general cosine values are in the $[-1, 1]$ range, however when using an LSI-index the cosine between its element never strays much below zero. This is since the LSI-index is derived from a term-document matrix that contains positive occurrence data only.

First matrix in Figure 4.3. To visualize similarities between documents we map them to gray values: the darker, the more similar. The similarities between elements are arranged in a square matrix called *correlation matrix* or *dot plot*. A correlation matrix is a common visualization tool to analyze patterns in a set of entities [126]. Each dot $a_{i,j}$ denotes the similarity between element d_i and element d_j. Put in other words, the elements are arranged on the diagonal and the dots in the off-diagonal show the relationship between them.

Figure 4.3: From left to right: unordered correlation matrix, then sorted by similarity, then grouped by clusters

4.2.3 Clustering: Ordering the Correlation Matrix

Without proper ordering the correlation matrix looks like television tuned to a dead channel. An unordered matrix does not reveal any patterns. An arbitrary ordering, such as for example by the names of the elements, is generally as useful as random ordering [24]. Therefore, we cluster the matrix to put similar elements near each other and dissimilar elements far apart.

A clustering algorithm groups similar elements together and aggregates them into clusters [97]. Hierarchical clustering creates a tree of nested clusters, called a *dendrogram*, which has two features: breaking the tree at a given threshold groups the elements into clusters, and traversing the tree imposes a sort order upon its leaves. We use these two features to rearrange the matrix and to group the dots into rectangular areas.

Second and third matrix in Figure 4.3. Each rectangle on the diagonal represents a lexical cluster: the size is given by the number of classes that belong to a topic, the color referring to the *semantic cohesion* [137] (that is the average similarity among its classes[4]). The color is the darker the more similar two clusters are, if it is white they are not similar at all. The position on the diagonal is ordered to make sure that similar topics are placed together.

The clustering takes the focus of the visualization from similarity between elements to similarity between clusters. The tradeoff is, as with any abstraction, that some valuable detail information is lost. Our experiments showed that one-to-many relationships between an element and an entire cluster are valuable patterns.

[4]Based on the similarity $\text{sim}(a, b)$ between elements, we define the similarity between cluster A and cluster B as $\frac{1}{|B|\times|A|}\sum\sum \text{sim}(a_m, b_n)$ with $a \in A$ and $b \in B$ and in the same way the similarity between an element a_0 and a cluster B as $\frac{1}{|B|}\sum \text{sim}(a_0, b_n)$ with $B \in B$.

4.3 Analyzing the Distribution of Lexical Clusters

The lexical clusters help us grasp the topics implemented in the source code. However, the clustering does not take the structure of the system into account. As such, an important question is: How are these topics distributed over the system?

To answer this question, we use a Distribution Map [182, 58]. A Distribution Map visualizes the distribution of properties over system parts that is a set of entities. In this chapter, we visualize packages and their classes, and color these classes according to the lexical cluster to which they belong.

For example, in Figure 4.4 we show an example of a Distribution Map representing 5 packages, 37 classes and 4 lexical clusters. Each package is represented by a rectangle, which includes classes represented as small squares. Each class is colored by the lexical cluster to which it belongs.

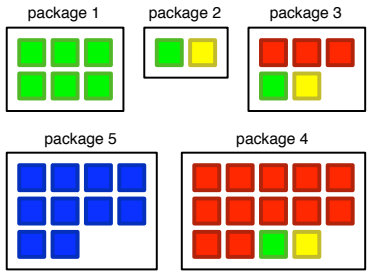


Figure 4.4: Example of a Distribution Map.

Using the Distribution Map visualization we correlate linguistic information with structural information. The lexical partition of a system, as obtained by lexical clustering, does generally not correspond one-on-one to its structural modularization. In most systems we find both topics that correspond to the structure as well as topics that cross-cut it. Applying this visualization to several case studies, we identified the following patterns:

- *Well-encapsulated topic* – if a topic corresponds to system parts, we call this a *well-encapsulated topic*. Such a topic is spread over one or multiple parts and includes almost all source code within those parts. If a well-encapsulated topic covers only one part we speak of a *solitary topic*.

- *Cross-Cutting topic* – if a topic is orthogonal to system parts, we call this a *cross-cutting topic*. Such a topic spreads across multiple parts, but includes only one or very few elements within each parts. As linguistic information and structure are independent of each other, cross-cutting identifiers or names do not constitute a design flaw. Whether a cross-cutting topic has to be considered a design smell or not depends on the particular

circumstances. Consider for example the popular three-tier architecture: It separates *accessing, processing* and *presenting data* into three layers; where application specific topics – such as for example *accounts, transactions* or *customers* – are deliberately designated to cross-cut the layers. That is, it emphasizes the separation of those three topics and deliberately designates the others as cross-cutting concerns.

- *Octopus topic* – if a topic dominates one part, as a solitary does, but also spreads across other parts, as a cross-cutter does, we call this an *octopus topic*. Consider for example a framework or a library: there is a core part with the implementation and scattered across other parts there is source code that plugs into the core, and hence uses the same vocabulary as the core.

- *Black Sheep topic* – if there is a topic that consists only of one or a few separate source documents, we call this a *black sheep*. Each black sheep deserves closer inspection, as these documents are sometimes a severe design smell. Yet as often, a black sheep is just an unrelated helper class and thus not similar enough to any other topic of the system.

4.4 Case studies

To show evidence of the usefulness of our approach for software comprehension, in this section we apply it to two case studies. First, we exemplify each step of the approach and discuss its findings in the case of JEdit, a text editor written in Java. This case study is presented in full length. Secondly, we present JBoss, an application-server written in Java, which includes interesting anomalies in its vocabulary.

Case Study	language	type	docs	terms	parts	rank	sim
Ant	Java	*Classes*	665	1787	9	17	0.4
Azureus	Java	*Classes*	2184	1980	14	22	0.4
JEdit	Java	*Classes*	394	1603	9	17	0.5
JBoss	Java	*Classes*	660	1379	10	16	0.5
Moose[5]	Smalltalk	*Classes*	726	11785	–	27	–
MSEModel	Smalltalk	*Methods*	4324	2600	–	32	0.75
Outsight	Java	*Classes*	223	774	10	12	0.5

Figure 4.5: The statistics of sample case studies, JEdit and JBoss are discussed in this work, for the other studies please refer to our previous work [109, 107].

Figure 4.5 summarizes the problem size of each case study. It lists for each case study: (lang) the language of the source code, (type) the granularity of documents, (docs) the number of documents, (terms) the number of terms, (parts) the number of found topics, (rank) the dimension of the LSI-index, and (sim) the threshold of the clustering.

[5]The Moose case study in [109] did not use stemming to preprocess the text corpus, hence the large vocabulary.

4.4.1 On the Calibration of Parameters and Thresholds

Our approach depends on several parameters, which may be difficult to choose for someone not familiar with the underlying technologies. In this section we present all parameters, discuss their calibration and share our experience gained when performing case studies using the Hapax tool.

Weighting the term-document-matrix. To balance out the influence of very rare and very common terms, it is common in information retrieval to weight the occurrence values. The most common weighting scheme is *tf-idf*, which we also use in the case studies, others being entropy or logarithmic weighting [62].

When experimenting with different weighting schemes, we observed that the choice of the weighting scheme has a considerable effect on the similarity values, depending on weighting the distance within the complete text corpus becomes more compact or more loose [145]. Depending on the choice of the weighting scheme, the similarity thresholds may differ significantly: as a rule of thumb, using logarithmic weighting and a similarity threshold of $\delta = 0.75$ is roughly equivalent to a threshold of $\delta = 0.5$ with *tf-idf* weighting [146].

Dimensionality of the LSI-space. As explained in Section 4.1, LSI replaces the term-document matrix with a low-rank approximation. When working with natural language text corpora that contain millions of documents and some tens of thousands of terms, most authors suggest to use an approximation between rank 200 and 500. In Software Analysis the number of documents is much smaller, such that even ranks as low as 10 or 25 dimensions yield valuable results. Our tool uses rank $r = (m * n)^{0.2}$ by default for an $m \times n$-dimensional text corpus, and allows customization.

Choice of clustering algorithm. There is a rich literature on different clustering algorithms [97]. We performed a series of experiments using different algorithms and we decided to use a hierarchical *average-linkage* clustering as it is a common standard algorithm. Further studies on the choice of clustering algorithm are open for future work.

Breaking the dendrogram into clusters. Hierarchical clustering uses a threshold to break the dendrogram, which is the tree of all possible clusters, into a fixed partition. Depending on the objective, we break it either into a fixed number of clusters (for example for the Distribution Map, where the number of colors is constrained) or at a given threshold (for example for the correlation matrix). In the user interface of the Hapax tool, there is a slider for the threshold such that we can immediately observe the effect on both correlation matrix and Distribution Map interactively.

4.4.2 Lexical Clustering applied on JEdit

We exemplify our approach using the case of JEdit, an open-source Text editor written in Java. The case study contains 394 classes and uses a vocabulary of 1603 distinct terms. We reduced the text corpus to an LSI-space with rank $r = 15$ and clustered it with a threshold of $\delta = 0.5$ (the choice of parameters is

discussed in subsection 4.4.1).

In Figure 4.6, we see nine clusters with a size of (from top right to bottom left) 116, 63, 26, 10, 68, 10, 12, 80, and 9 classes. The system is divided into four zones: (zone 1) the large cluster in the top left, (zone 2) two medium sized and one small clusters, (zone 3) a large cluster and two small clusters, and (zone 4) a large and a small cluster. The two zones in the middle are both similar to the first zone but not to each other, and the fourth zone is not similar to any zone.

In fact, there is a limited area of similarity between the Zone 2 and 3. We will later on identify the two counterparts as topics Pink and Cyan, which are related to text buffers and regular expression respectively. These two topics share some of their labels (that is start, end, length and count), however they are clustered separately since LSI does more than just keyword matching. LSI is taking the context of term usage into account as well, that is the co-location of terms with other terms.

This is a common pattern that we often encountered during our experiments: zone 1 is the core of system with domain-specific implementation, zone 2 and 3 are facilities closely related to the core, and zone 4 is an unrelated component or even a third-party library. However, so far this is just an educated guess and therefore we will have a look at the labels next.

Figure 4.6 lists for each cluster the top-7 most relevant labels, ordered by relevance. The labels provide a good description of the clusters and tell same story as the correlation matrix before. We verified the labels and topics by looking at the actual classes within each cluster.

- Zone 1: topic Red implements the very domain of the system: files and users, and a user can load, edit and save files.
- Zone 2: topic Green and Magenta implement the user interface, and topic Pink implements text buffers.
- Zone 3: topic Cyan is about regular expressions, topic Yellow provides XML support and topic DarkGreen is about TAR archives.
- Zone 4: topic Blue and Orange are the BeanShell scripting framework, a third-party library.

All these labels are terms taken from the vocabulary of the source code and as such they do not always describe the topics in generic terms. For example, event though JEdit is a text-editor, the term *text-editor* is not used on the source code level. The same applies for topic Cyan, where the term *regular expression* does not show up in the labels.

Figure 4.7 shows the distribution of topics over the package structure of JEdit. The large boxes are the packages (the text above is the package name), the squares are classes and the colors correspond to topics (the colors are the same as on Figure 4.6).

For example, in Figure 4.7 the large box on the right represents the package named *bsh*, containing over 80 classes most of which implement the topic referred

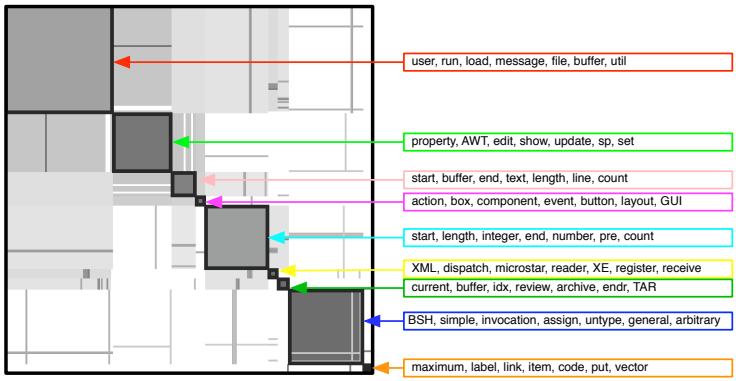


Figure 4.6: The lexical clusters of JEdit and their labels.

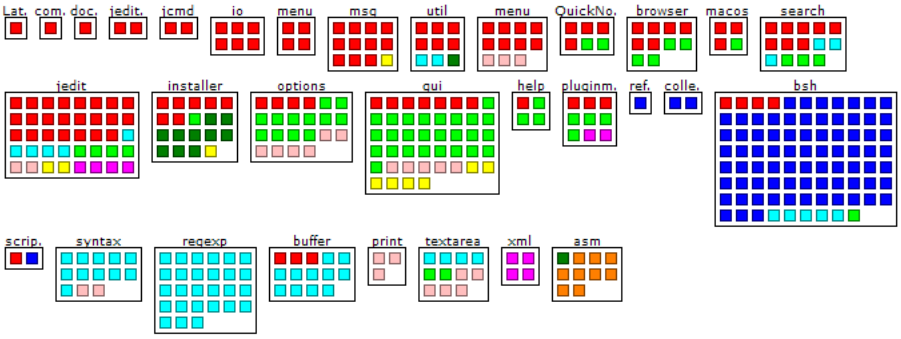


Figure 4.7: The Distribution Map of the lexical clusters over the package structure of JEdit.

to by Blue. The package boxes are ordered by their similarity, so that related packages are placed near to each other.

Topic Red, the largest cluster, shows which parts of the system belong to the core and which do not. Based on the ordering of the packages, we can conclude that the two UI topics (Green and Yellow) are more closely related to the core than for example topic Cyan, which implements regular expressions.

The three most well-encapsulated topics (for example Orange, Blue and Cyan) implement separate topics such as scripting and regular expressions. Topic Yellow and Pink cross-cut the system: Yellow implements dockable windows, a custom GUI-feature, and Pink is about handling text buffers. These two topics are good candidates for closer inspection, since we might want to refactor them into packages of their own.

4.4.3 First Impression of JBoss: Distribution Map and Labels

This case study presents the outline of JBoss, an application-server written in Java. We applied lexical clustering and partitioned the system into ten topics. The system is divided into one large cluster (colored in red), which implements the core of the server, and nine smaller clusters. Most of the small clusters implement different services and protocols provided by the application server.

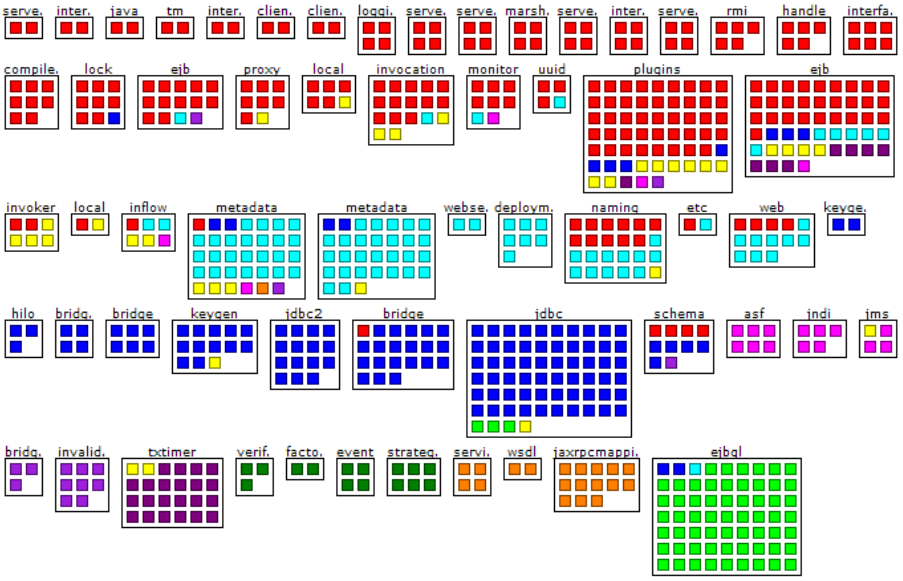


Figure 4.8: The Distribution Map of the linguistic clusters over the package structure of JBoss.

Color	Size	Labels
red	223	invocation, invoke, wire, interceptor, call, chain, proxy, share
blue	141	jdbccmp, JDBC, cmp, field, loubyansky, table, fetch
cyan	97	app, web, deploy, undeployed, enc, JAR, servlet
green	63	datetime, parenthesis, arithmetic, negative, mult, div, AST
yellow	35	security, authenticate, subject, realm, made, principle, sec
dark magenta	30	expire, apr, timer, txtimer, duration, recreation, elapsed
magenta	20	ASF, alive, topic, mq, dlq, consume, letter
orange	20	qname, anonymous, jaxrpcmap, aux, xb, xmln, WSDL
purple	16	invalid, cost, September, subscribe, emitt, asynchron, IG
dark green	15	verify, license, warranty, foundation, USA, lesser, fit

Figure 4.9: The labels of the lexicel clusters of JBoss.

The Distribution Map is illustrated in Figure 4.8, and the top-7 labels are listed in figure Figure 4.9 in order of relevance. We verified the clustering by looking the source code, and present the results as follows.

Topic Red is the largest cluster and implements the core functionality of the system labeled with terms such as *invocation, interceptor, proxy* and *share*. Related to that, topic Cyan implements the deployment of JAR archives.

The most well-encapsulated topics are DarkGreen, Orange, Green and Blue. The first three are placed apart from Red, whereas Blue has outliers in the red core packages. The labels and package names (which are printed above the package boxes in the Distribution Map) show that DarkGreen is a bean verifier, that Orange implements JAX-RPC and WDSL (for example web-services), that Green implements an SQL parser and that Blue provides JDBC (for example database access) support. These are all important topics of an application server.

The most cross-cutting topic is Yellow speading across half of the system. The labels reveal that this is the security aspect of JBoss, which is reasonable as security is an important feature within a server architecture.

Noteworthy is the label *loubyansky* in topic Blue, it being the name of a developer. Based on the fact that his name appears as one of the labels, we assume that he is the main developer of that part of the system. Further investigation proved this to be true.

Noteworthy as well are the labels of topic DarkGreen, as they expose a failure in the preprocessing of the input data. To exclude copyright disclaimers, as for example the GPL licence, we ignore any comment above the *package* statement of a Java class. In the case of topic DarkGreen this heuristic failed: the source files contained another licence within the body of the class. However, repeating the same case study with an improved preprocessing resulted in nearly the same clustering and labeled this cluster as RMI component: *event, receiver, RMI, RMIiop, iiop, RMIidl, and idl.*

The topics extracted from the source code can help improving comprehension. If a maintainer is seeking information, lexical clustering helps in identifying the related code. This is similar to the use of a search engine, for example if the web-service interface has to be changed, the maintainer can immediately look at the Orange concept, and identify the related classes. Much in the same way, to maintain the database interface a developer may look at the Blue concept.

4.5 Discussion

In this section we evaluate and discuss success criteria, strengths and limitations of the proposed approach. We discuss how the approach depends on the quality of the identifer naming. Furthermore we discuss the relation between linguistic topics and domain or application concepts.

4.5.1 On the Quality of Identifier Names

In the same way as structural analysis depends on correct syntax, lexical analysis is sensitive to the quality of the naming. Since we derive our topics solely based

on the use of identifer names and comments, it does not come as a surprise that our approach depends on the quality of the source code naming.

Our results are not generalizable to arbitrary software systems. Good naming convention and well chosen identifiers yields best results, whereas bad naming (that is too generic names, arbitrary names or cryptic abbreviations) is one of the main threats to external validation. The vocabulary of the case studies presented in this work is of good quality, however, when performing other case studies we learned of different facets that affect the outcome, these being:

On the use of naming conventions. Source following state-of-the-art naming conventions, as for example the Java Naming Convention, is easy to preprocess. In case of legacy code that uses other naming conventions (for example the famous Hungarian Notation) or even none at all, other algorithms and heuristics are to be applied [42, 4].

On generic or arbitrarily named identifiers. However, even the best preprocessing cannot guess the meaning of variables which are just named *temp* or *a*, *b* and *c*. If the developers did not name the identifiers with care, our approach fails, since the developer knowledge is missing. Due to the strength of LSI in detecting synonymy and polysemy, our approach can deal with a certain amount of such ambiguous or even completely wrongly named identifiers – but if a majority of identifiers in a system is badly chosen, the approach fails.

On abbreviated identifier names. Abbreviated identifers are commonly found in legacy code, since early programming languages often restrict the discrimination of identifer names to the first few letters. But unlike generic names, abbreviations affect the labeling only and do not threaten our approach as whole. This might come as a surprise, but since LSI is solely based on analyzing the statistical distribution of terms across the document set, it is not relevant whether identifiers are consistently written out or consistently abbreviated.

However, if the labeling task comes up with terms such as *pma*, *tcm*, *IPFWDIF* or *sccpsn* this does not tell a human reader much about the system. These terms are examples taken from a large industry case study, which is not included in this chapter, where about a third of all identifiers were abbreviations. In this case a more useful labeling can be achieved using approaches that recover abbreviations [4].

On the size of the vocabulary. The vocabulary of source code is very small, smaller than that of a natural language text corpus. Intuitively explained: LSI is like a child learning language. In the same way as a human with a vocabulary of 2000 terms is less eloquent and knowledgeable than a human with a vocabulary of 20,000 terms, LSI performs better the larger the vocabulary. The smaller the vocabulary the stronger the effect of missing or incorrect terms. In fact, LSI has been proven to be a valid model of the way children acquire language [117].

On the size of documents. On average there are only about 5-10 distinct terms per method body, and 20-50 distinct terms per class. In a well commented software system, these numbers are higher since comments are human-readable text. This is one of the rationales why LSI does not perform as accurately on source code as on natural language text [47], however the results are of sufficient

quality.

On the combination of LSI with morphological analysis. Even though the benefit of stemming (that is removing the morphological suffix of words) is not without controversy, we apply it as part of the preprocessing step [13]. Our rationale is: analyzing a software system at the level of methods is very sensitive to the quality of input, as the small document size threatens the success of LSI. Considering these circumstances, we decided to rely on stemming as it is well known that the naming of identifers often includes the same term in singular and plurals: for example *setProperty* and *getAllProperties* or *addChild* and *getChildren*.

4.5.2 On using Lexical Clustering for Topic Identification

One of our objectives is to compare linguistic topics to domain and application concepts [29]. We derive linguistic topics from the vocabulary usage of source code instead from external definitions. In this section we clarify some questions concerning the relation between derived topics and externally defined concepts:

On missing vocabulary and ontologies. Often the externally defined concepts are not captured by the labeling. The rationale for this is as follows. Consider for example a text editor in whose source code the term *text-editor* is never actually used, but terms like *file* and *user*. In this case our approach will label the text-editor concepts with these two terms, as a more generic term is missing. As our approach is not based on an ontological database, its vocabulary is limited to the terms found in source code and if terms are not used our approach will not find accurate labels. We suggest to use ontologies (that is WordNet) to improve the results in these cases.

On the congruence between topics and domain. When starting this work, one of our hypotheses was that lexical clustering will reveal a system's domain semantics. But our experiments disproved this hypothesis: most linguistic topics are application concepts or architectural components, such as layers. In many experiments, our approach partitioned the system into one (or sometimes two) large domain-specific part and up to a dozen domain-independent parts, such as for example input/output or data storage facilities. Consider for example the application in Figure 4.10, which is divided into nine parts as follows:

Only one topic out of nine concepts is about the system's domain: job exchange. Topic Red includes the complete domain of the system: that is users, companies and CVs. All other topics are application specific components: topic Blue is a CV search engine, topic DarkGreen implements PDF generation, topic Green is text and file handling, topic Cyan and Magenta provide access to the database, and topic DarkCyan is a testing and debugging facility. Additionally the cross-cutting topic Yellow bundles high-level clones related to time and timestamps.

On the congruence between topics and packages. In section Section 4.3 we discussed the relation between topics and packages. Considering again the case study in Figure 4.10 as an example, we find occurrences of all four patterns: Topic DarkGreen for example is well-encapsulated, whereas topic

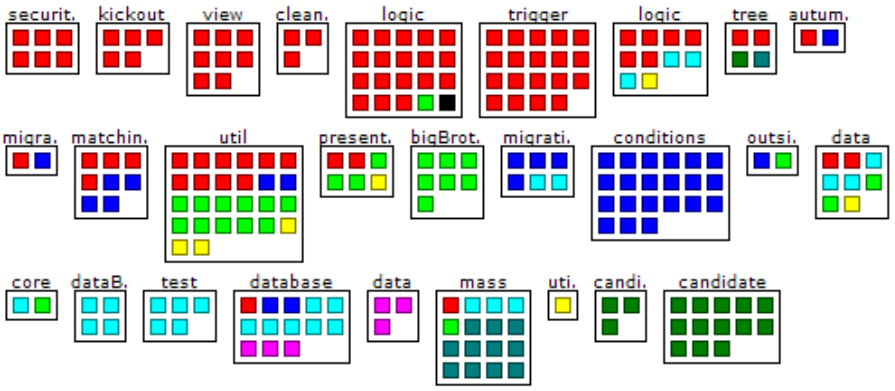


Figure 4.10: The Distribution Map of Outsight, a web-based job portal application [107].

Yellow cross-cuts the application. Then there is topic Blue, which is an octopus with the *conditions* package as its body and tentacles reaching into six other packages, and finally we have in the *logic* package an instance of a black sheep.

On source documents that are related to more than one topic. If we want to analyze how the topics are spread across some type of documents (for example packages, classes or methods) we have to break the system into documents one level below the target level. For example, if we want to analyze the topic distribution over packages, we break the system into classes and analyze how the topics of classes are spread over the packages.

Chapter 5

Code Summarization

Aim	Refinding and discovering topics.
Reach	Part of a system's codebase or history.
Clues	Lexcial and episodic (established through lexical information).
Query	Visual analytics of word clouds.

Source code rarely comes with an executive summary. When developers encounter a piece of source code for the first time they are typically not presented with a high-level summary of the code's topics. Looking at package and class names might help to make an educated guess but does often not give away the complete picture. Furthermore, when comparing two pieces of source code or when comparing two versions of the same system, developers are missing a high-level summary that distills the difference in a short textual summarization. The need for software summarization is obvious.

In the following chapter we present an approach to summarize source code as a word cloud. When summarizing part of a system the word cloud consists of the statistically most significant terms that set this part of system apart from the rest. The same approach can be use to compare two parts of a system or to compare two versions of the same system. When comparing version, both statistically significant additions and removals are shown using two different colors. Presenting those word clouds to the developer helps them to lexically query the topics, as well as when presenting all words clouds of a system's versions to recover and tell the story of a system's history and thus enabling developers to draw from episodic memories that they possibly never experienced first-hand themselves.

akuhn ►*Include MSR Poster*◄

In recent years, lexical information found in source code has proven to be a valuable source for software analysis, often including the retrieval of labels [18, 89, 110]. However, labeling software is not without pitfalls. The distribution of words in software corpora follows the same power-law as word frequencies in natural-language text [127]. Most of the text is made up of a small set of

common terms, whereas content-bearing words are rare. Analysis of software vocabulary must deal with the reality of rare terms, thus statistical tests that assume normal distribution are not applicable. For example, textual comparison based on directly counting term frequencies is subject to overestimation when the frequencies involved are very small.

Our approach uses log-likelihood ratios of word frequencies to automatically provide labels for software components. We present a prototype implementation of our labeling/comparison algorithm and provide examples of its application. In particular, we apply the approach to detect trends in the evolution of a software system. For text analysis the use of log-likelihood ratio improves the statistical results. Likelihood tests do not depend on assumptions of normal distribution, instead they use the asymptotic distribution of binomial likelihood [64]. Using log-likelihood ratios allows comparisons to be made between the significance of occurrences of both common and rare terms.

The approach that we present can be applied i) to compare components with each other, ii) to compare a component against a normative corpus, and iii) to compare different revisions of the same component. We present a prototype implementation and give examples of its application. In particular, we apply the approach to detect trends in the evolution of the JUnit software system.

5.1 Log-Likelihood in a Nutshell

This section explains how log-likelihood ratio is applied to analyse word frequencies. The explanations are kept as concise as possible. We provide the general background and just enough details such that a programmer may implement the algorithm. Please refer to Ted Dunning's work [64] for more background.

The idea behind log-likelihood ratio is to compare two statistical hypotheses, of which one is a subspace of the other. Given two text corpora, we compare the hypothesis that both corpora have the same distribution of term frequencies with the "hypothesis" given by the actual term frequencies. Because we know that terms are not equally distributed over source code, we use binomial likelihood

$$L(p,k,n) = p^k(1-p)^{n-k}$$

with $p = \frac{k}{n}$, where k is the term frequency (that is number of occurrences) and n the size of the corpus. Taking the logarithm of the likelihood ratio gives [1]

$$\begin{aligned} -2\log\lambda = \; & 2\big[\log L(p_1,k_1,n_1) + \log L(p_2,k_2,n_2) \\ & - \log L(p,k_1,n_1) - \log L(p,k_2,n_2)\big] \end{aligned}$$

[1] In some publications (for example [160]) we found that the last two terms were omitted (since their values tend to be orders of magnitude smaller than the corresponding values of the first two terms). The results presented in this chapter, however, are obtained using the complete log likelihood ratio formula.

with $p = \frac{k_1+k_2}{n_1+n_2}$. The higher the value of $-2log\lambda$ the more significant is the difference between the term frequencies in of both text corpora. By multiplying the $-2log\lambda$ value with the signum of $p_1 - p_2$ we can further distinguish between terms specific to the first corpus and terms specific to the second corpus. Terms that are equally frequent in both corpora have a $-2log\lambda$ value close to zero and thus fall in between.

Example. Let C_1 be the corpus of a software project with size $n_1 = 10^6$, where the words `'rare'`, `'medium'`, and `'common'` appear respectively 1, 100, and 1×10^4 times; and let C_2 be the corpus of one of the project's classes with size $n_2 = 1000$, where each word appears 10 times. Then the log-likelihood ratio values are

	p_1	p_2	$-2log\lambda$	χ^2
`rare`	10^{-6}	10^{-2}	131.58	9.08
`medium`	10^{-4}	10^{-2}	71.45	0.89
`common`	10^{-2}	10^{-2}	0.00	0.00

The column χ^2 lists the value of Pearson's chi-square test, which assumes normal distribution. As we can see, there is an overestimation when the frequencies involved are very small. Therefore, text analysis should use log-likelihood ratios to compare the occurrences of common and rare terms [64].

5.2 Applications

In this section we present two example applications of log-likelihood ratio for software analysis. There are two main types of corpus comparison: comparison of a sample corpus to a larger corpus, and comparison of a two equally sized corpora. In the first case, we refer to the large corpus as a *normative* corpus since it provides as norm against which we compare.

Applications of these comparisons are

- *Providing labels for components.* Comparing a component's vocabulary with a large normative corpus (as for example Sourceforge, Github, or Sourcerer [17]), we obtain labels that describe the component. In the same way, we can compare a class's vocabulary against the containing project.
- *Comparing components to each other.* Comparing two components, we obtain labels to describe their differences as well as commonalities. This is applicable at any level of granularity, from the level of projects down to the level of methods.
- *Documenting the history of a component.* Comparing subsequent revisions of the same component, we obtain labels to describe the evolution of that component. (Using multinominal distribution we could even compare all revisions at once, although such results are harder to interpret [64].)

java.io		java.text		java.util	
read	521.99	pattern	228.92	iterator	306.91
write	481.93	format	209.40	entry	301.90
skip	154.61	digits	183.24	next	237.82
close	113.41	FIELD	167.58	E	222.33
mark	111.47	instance	127.16	contains	187.69
println	99.66	fraction	104.98	sub	166.49
UTF	85.27	integer	102.77	of	165.57
flush	80.96	index	93.43	K	154.07
desc	69.19	run	90.99	T	154.30
TC	68.88	currency	91.55	key	145.10
prim	61.48	decimal	86.34	all	145.74
char	61.28	contract	84.92	V	142.15
buf	60.15	separator	72.11	remove	128.59
stream	56.86	grouping	62.26	last	128.09
fields	52.38	parse	56.93	map	115.68
bytes	47.99	collation	56.60	clear	114.03
...		...		...	
border	-28.35	UI	-23.07	create	-62.67
set	-33.89	border	-22.77	listener	-62.42
remove	-37.49	property	-24.23	action	-63.83
listener	-39.79	remove	-30.11	UI	-64.68
accessible	-40.97	accessible	-32.91	border	-63.83
paint	-44.90	listener	-31.97	accessible	-92.25
value	-59.10	type	-32.18	paint	-101.11
get	-64.38	paint	-36.07	get	-164.14

Table 5.1: Labels retrieved for three Java packages using the full Java 6.0 API as normative corpus.

- *Describing the results of software search.* Code-search engines have recently received much attention, both commercial engines (as for example Krugle) and academic engines (as for example Sourcerer [17]) are publicly available. The result of a search query is typically provided by presenting a peep-hole view on the matching source line and its context to the user. Comparing each result, or the class/project that contains the result, against the entire index we can provide labels that may help users to make better use of these results.

- *Analysing the structural use of vocabulary.* There has been much confusion regarding which parts of the software vocabulary are to be considered in software analysis. Some approaches consider the entire source code including comments, keywords and literals (for example [115, 110]), other approaches consider class and methods names only (for example [18, 89]). A recent study compared the vocabulary of different structural levels using techniques that assume normal distribution [127]. Log-likelihood ratios provide a statistically more sound means to study these phenomena.

5.2.1 Labeling the Java API

In this example, we compare the packages `java.io` and `java.text` and `java.util` with the normative corpus of the full Java 6.0 API. We use the Java Compiler (JSR 199) to parse the byte-code of the full Java API and then extract the vocabulary of all public and protected elements. We extract the names of pack-

ages, classes (including interfaces, annotations, and enums), fields, methods and type parameters. We split the extracted names by camel-case to accomodate to the Java naming convention.

Results are shown in Table 5.1. For each package we list the most specific words and the least specific words. All three packages are characterized by not covering UI code, in addition `java.io` and `java.util` have obviously substantially fewer `get`-accessors than is usual for the Java API. The remaining findings offer no further surprises, except maybe for the uppercase letters in `java.util` which are generic type parameters; obviously the majority of the Java 6.0 API makes less use of generic types than the collection framework.

5.2.2 The Evolution of JUnit

In this example, we report on the vocabulary trends in the history of the `JUnit`[2] project. We use a collection of 14 release distributions of JUnit and parse the source code of each release. We compare the vocabulary of each two subsequent releases and report on the most significant changes in the vocabulary. We extract all words, including comments; split by camel-case, and exclude English stopwords but not Java keywords.

Results are shown in Table 5.2. For each release we list the top removed terms and the top added terms, as well as the $-2log\lambda$ value of the top-most term. Large $-2log\lambda$ values indicate substantial changes.

The top 7 change trends (that is $-2log\lambda \geqslant 100.0$) in the history of JUnit are as follows. In 3.2 removal of `MoneyBag` example and introduction of graphical UI; in 4.0 removal of graphical UI and introduction of annotation processing; in 4.2 removal of HTML tags from Javadoc comments; in 4.4 introduction of theory matchers and `hamcrest` framework; in 4.5 introduction of blocks and statements. We manually verified these findings with the release notes of JUnit.

[2] `http://www.junit.org`

JUnit	$2log\lambda$	**Top-10 terms (with $-2log\lambda \geqslant 10.0$)**
3	-8.21	
	54.11	count, writer, wrapper
3.2	-382.80	money, CHF, assert, case, USD, equals, test, fmb, result, currency
	114.21	tree, model, constraints, combo, reload, swing, icon, pane, browser, text
3.4	-19.48	stack, util, button, mouse
	15.73	preferences, base, zip, data, awtui
3.5	-38.78	param, reload, constraints
	69.34	view, collector, context, left, cancel, values, selector, views, icons, display
3.6	-1.20	
	8.72	
3.7	-8.25	
	2.79	
3.8	-13.30	deprecated
	23.40	printer, boo, lines
4.0	-349.34	constraints, grid, bag, set, label, panel, path, icon, model, button
	350.47	description, code, nbsp, org, annotation, notifier, method, request, runner, br
4.1	-1.43	
	61.90	link, param, check
4.2	-288.53	nbsp, br
	20.03	link, builder, pre, li
4.3.1	-8.91	
	53.36	array, actuals, expecteds, multi, dimensional, arrays, values, javadoc
4.4	-34.32	introspector, code, todo, multi, javadoc, dimensional, array, runner, test, fix
	151.98	matcher, theories, experimental, hamcrest, matchers, theory, potential, item, supplier, parameter
4.5	-30.11	theory, theories, date, result, static, validator, pointer, string, assert, experimental
	124.28	statement, model, builder, assignment, block, errors, unassigned, evaluate, describable, statements

Table 5.2: Evolution of JUnit: for each release we list the removed and the added words, large $2log\lambda$ values indicate more significant changes.

Chapter 6

Software Cartography

Aim	Code orientation in general.
Reach	Local codebase and possibly history of a system.
Clues	Spatial (established through lexical and structural information).
Query	Visual analytics of a cartographic visualization.

Current tool support for code orientation by spatial clues is *ad hoc* at best, most striking is the lack of spatial on-screen representations of source code. Without such a representation developers are barely able to draw on the strong spatial capability of the human brain.

We provide a cartographic on-screen visualization such that developers can start using spatial clues for code orientation. Since software has no inherent spatial structure we use lexical and structural information found in the source code to establish a spatial layout of the local code base. The created *software maps* are stable over time and can be shared among members of a team to establish a common mental model of the system. We implemented our approach in a prototype, evaluated it in a user study, and found that it is most helpful for spatially exploring search results and call hierarchies.

Software visualization can provide a concise overview of a complex software system. Unfortunately, since software has no physical shape, there is no "natural" mapping of software to a two-dimensional space. As a consequence most visualizations tend to use a layout in which position and distance have no meaning, and consequently layout typically diverges from one visualization to another. We propose an approach to consistent layout for software visualization, called *Software Cartography*, in which the position of a software artifact reflects its *vocabulary*, and distance corresponds to similarity of vocabulary. We use a vector-space model to map software artifacts to a vector space, and then a combination of the Isomap algorithm [179] and Multidimensional Scaling [33] is used to map this vector space down to two dimensions. The resulting consistent layout allows us to develop a variety of thematic *software maps* that express very different aspects of software while making it easy to compare them. The

approach is especially suitable for comparing views of evolving software, since the vocabulary of software artifacts tends to be stable over time. We present a prototype implementation of Software Cartography, and illustrate its use with practical examples from numerous open source case studies.

6.1 Spatial Representation of Software

Software visualization offers an attractive means to abstract from the complexity of large software systems. A single graphic can convey a great deal of information about various aspects of a complex software system, such as its structure, the degree of coupling and cohesion, growth patterns, defect rates, and so on [57, 103, 162, 176]. Unfortunately, the great wealth of different visualizations that have been developed to abstract away from the complexity of software has led to yet another source of complexity: it is hard to compare different visualizations of the same software system and correlate the information they present.

We can contrast this situation with that of conventional thematic maps found in an atlas. Different phenomena, ranging from population density to industry sectors, birth rate, or even flow of trade, are all displayed and expressed using *the same consistent layout*. It easy to correlate different kinds of information concerning the same geographical entities because they are generally presented using the same kind of layout. This is possible because (i) there is a natural mapping of position and distance information to a two-dimensional layout[1], and (ii) because by convention North is normally considered to be on the top.[2]

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

Robert DeLine's work on software navigation [50, 51] closely relates to Software Cartography. His work is based on the observation that developers are consistently lost in code [52] and that using textual landmarks only places a large burden on cognitive memory. He concludes the need for new visualization

[1] Even if we consider that the Earth is not flat on a global scale, there is still a natural mapping of position and distance to a two-dimensional layout; see the many types of cartographic projections (for example the Mercator projection) used during centuries to do that. In fact, this is true for a large class of manifolds.

[2] The orientation of modern world maps, that is North on the top, has not always been the prevailing convention. On traditional Muslim world maps, for example, South used to be in the top. Hence, if Europe would have fallen to the Ottomans at the Battle of Vienna in 1683, all our maps might be drawn upside down by now [88].

techniques that allow developers to use their spatial memory while navigating source code.

DeLine proposes four desiderata [50] 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 such 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.

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 lexical clues.
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.

6.2 Software Cartography

Consistent layout for software would make it easier to compare visualizations of different kinds of information. But what should be the basis for positioning representations of software artifacts within a "cartographic" software map? What we need is a semantically meaningful notion of position and distance for software artifacts, a spatial representation of software in a multi-dimensional space, which can then be mapped to consistent layout on the 2-dimensional visualization plane.

We propose to use *vocabulary* as the most natural analogue of physical position for software artifacts, and to map these positions to a two-dimensional space as a way to achieve consistent layout for software maps. Distance between software artifacts then corresponds to distance in their vocabulary. We use a combination of the Isomap algorithm [179] and Multidimensional Scaling [33] to projection the high-dimensional vector space model onto the two-dimensional visualization pane. Finally we use cartographic techniques (such as digital elevation, hill-shading and contour lines) to generate a landscape representing the frequency of topics. We call our approach *Software Cartography*, and call a

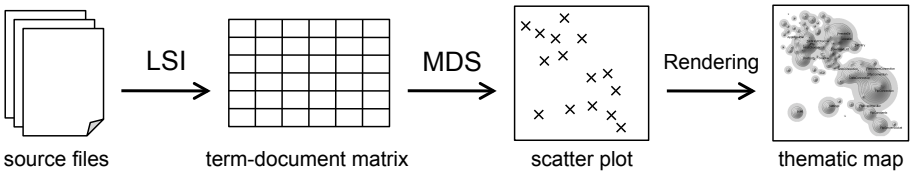


Figure 6.1: Software Cartography in a nutshell: left) the raw text of source files are parsed and indexed, center) the high-dimensional term-document-matrix is mapped down to two dimensions using Multidimensional Scaling and Isomap, and right) cartographic visualization techniques are used to render the final map.

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

Why should we adopt vocabulary as distance metric, and not some structural property? First of all, vocabulary can effectively *abstract* away from the technical details of source code [110] by capturing the key domain concepts of source code. Software entities with similar vocabulary are conceptually and topically close. Lexical similarity has proven useful to detect high-level clones [136] and cross-cutting concerns [18] in software. Furthermore, it is known that over time vocabulary tends to be more stable than the structure of software [8], and tends to grow rather than to change [186]. Although refactorings may cause functionality to be renamed or moved, the overall vocabulary tends not to change, except as a side-effect of growth. This suggests that vocabulary will be relatively *stable* in the face of change, except where significant growth occurs. As a consequence, vocabulary not only offers an intuitive notion of position that can be used to provide a consistent layout for different kinds of thematic maps, but it also provides a robust and consistent layout for mapping an evolving system. System growth can be clearly positioned with respect to old and more stable parts of the same system.

In the following 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 6.1, is as follows:

Information Retrieval 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-Dimensional Embedding A metric distance is used to compute the pairwise dissimilarity of software artifacts (typically source code files). A combination of the Isomap algorithm [179] and Multidimensional Scaling [33] is used to embed all software artifacts on the visualization pane. Other than with Lexical Clustering, as presented in Chapter 4, we do not apply Latent Semantic Indexing as it has been found to have little impact on the

final embedding, if at all. The application of Isomap is an improvement of the embedding in order to assist MDS with the global layout. The final embedding minimizes the error between the dissimilarity values and the visual distances.

Early prototypes of CODEMAP used a distance metric that was based on lexical similarity only. However, our user study revealed that developers tend to interpret visual distance as a measure of structural dependencies, even though they were aware of the underlying lexical implementation. Based on this observation, we developed an improved distance metric that takes both lexical similarity and structural distance (based on the "Law of Demeter" [125]) into account.

Digital Elevation Model In the next step, a digital elevation model is created. Each software artifact contributes a Gaussian shaped basis function to the elevation model according to its KLOC size. The contributions of all software artifacts are summed up and normalized.

Cartographic rendering In the final step, hill-shading is used to render the landscape of the software map. Please refer to previous work for full details [115]. Metrics and markers are rendered in transparent layers on top of the landscape. Users can toggle separate layer on/off and thus customize the codemap display to their needs.

We implemented a prototype of our approach, CODEMAP, which is available as an open source project. CODEMAP was originally programmed in Smalltalk, in the mean time development has been moved to Java. CODEMAP is available as an Eclipse plug-in[3].

6.2.1 Lexical Similarity between Source Files

As motivated in the introduction, the distance between software entities on the map is based on the lexical similarity of source files. 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 (that is identifier 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 [196]. 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

[3] http://scg.unibe.ch/codemap

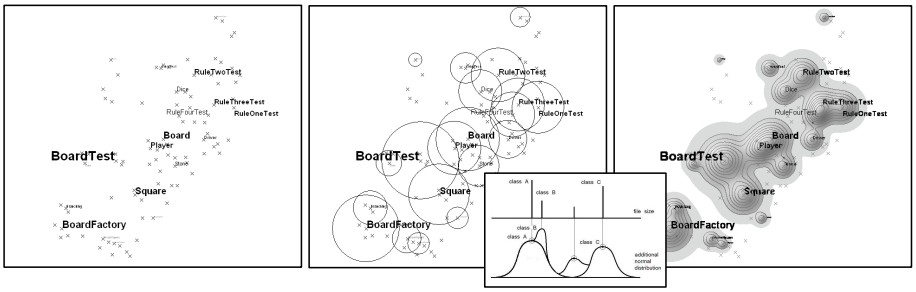


Figure 6.2: Construction steps: left) the projection step placement of files on the visualization pane, middle) circles around each file's location, based on class size in KLOC, right) digital elevation model with hill-shading and contour lines. Sidebox on digital elevation model) each file contributes a Gaussian shaped basis function to the elevation model according to its KLOC size. The contributions of all files are summed up to a digital elevation model.

convention which is predominantly used in Java source code. Note that since our approach is based on raw text, any programming language that uses textual source files might be processed.

6.2.2 Multi-dimensional Scaling

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 a combination of the Isomap algorithm [179] and Multidimensional Scaling [33] in order to map from the previously established multi-dimensional term-document matrix down to two dimensions. Multidimensional Scaling tries to minimize a stress function while iteratively placing elements into a low-level space. Multidimensional Scaling yields the best approximation of a vector space's orientation, that is it preserves the distance relation between elements as best as possible. This is good for data exploration problems.

6.2.3 Cartographic Visualization Techniques

Eventually, we use hill-shading [39] to render an aesthetically appealing landscape. Figure 6.2 illustrates the final rendering steps of Software Cartography. On the final map, each source file is rendered as a hill whose height corresponds to the entity's KLOC size.

Hill-shading uses a digital elevation model (DEM) to render the illumination of a landscape. The digital elevation model is a two-dimensional scalar field. Each each entity contributes a Gaussian shaped basis function to the el-

evation model. To avoid that closely located entities occlude each other, the contributions of all files are summed up as shown in Figure 6.2.

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, as an algorithm is needed to ensure that labels do not overlap. Also labels should not obscure important landmarks. Most labeling approaches are semi-automatic and need manual adjustment, an optimal labeling algorithm does not exist [172].

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. This work uses a greedy brute-force algorithm for labeling. Labels are placed in order of hill size, that is the name of the largest file is placed first, and so on. If a to-be placed label would overlap with an already placed label, the to-be placed label is omitted. Thus, the labels of smaller files are typically omitted in favor of the labels of larger files.

6.3 On the Choice of Vocabulary

The decision to use a distance based on lexical similarity does, indeed, create a distribution of distances that should not change a lot in time. This is because programmers will not use a completely new set of lexical tokens in each new version of the software. In fact, it has been shown that over time vocabulary tends to be more stable than the structure of software [8]. However, this also will create software maps that naturally only can show how items are similar from a lexical point of view.

The map layout as presented in this work can, of course, be used to see how items are related from the point of view of some other distance metric, such as considering structural similarity, similarity with regard to a complexity or testability metric. In that case, the distance may vary a lot over time during the evolution of a product, and this will create unstable layouts. The focus of this work, however, is the creation of maps that help programmers to establish a stable mental model of their software system under work. In any case, if maps based on other metrics are ever to be used in conjunction with vocabulary-based Software Cartography maps, we strongly recommend to visually distinguish them by using another rendering scheme. This helps to reduce the likeliness that programmers confuse the spatial layout of these other maps, with the mental model acquired through the use of Software Cartography maps.

As mentioned above, Software Cartography is vocabulary-based because vocabulary can effectively *abstract* away from the technical details of source code [110] by capturing the key domain concepts of source code. The assumption is that software entities with similar vocabulary are conceptually and topically close. Consider, for example, programming languages and software where methods may be overloaded. Even though overloaded methods differ in their implementation strategy, they will typically implement the same concept using the

same vocabulary. In fact, lexical similarity has proven useful to detect high-level clones [136] and cross-cutting concerns [18] in software.

Although refactorings may cause functionality to be renamed or moved, the overall vocabulary tends not to change, except as a side-effect of growth [196, 186]. Consider the example of a rename refactoring. Two effects may occur. In the first case, all occurrences of a symbol are replaced with new symbol. This will not affect the map, since lexical similarity is based on statistical analysis only. Replacing all occurrences of one term with a new term is, from the point of these IR 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 will constitute a better representation of the system. On the other hand, if the new name is a bad choice, the new layout will be 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 [90].

Not considered in the present work is the relative weight of different lexical tokens. For example, it seems reasonable to weight local identifiers differently than identifiers in top-level namespaces. Also, one may treat names coming from library functions different from the ones coming from the actual user code. Given the absence of evaluation benchmarks, we decided to use equal weighting for all lexical tokens. Also, preliminary experiments with different weighting schemes indicate that relative weights below boost level, that is below a factor of 10, do often not significantly affect the overall layout.

6.4 Example: the Story of Ludo

In this section we present an example of Software Cartography. Figure 6.3 shows the complete history of the Ludo system, consisting of four iterations. Ludo is used in a first year programming course to teach iterative development. The 4th iteration is the largest with 30 classes and a total size of 3-4 KLOC. We selected Ludo because in each iteration, a crucial part of the final system is added.

- The first map (Figure 6.3, leftmost) shows the initial prototype. This iteration implements the board as a linked list of squares. Most classes are located in the south-western quadrant. The remaining space is occupied by ocean, nothing else having been implemented so far.

- In the second iteration (Figure 6.3, second to the left) the board class is extended with a factory class. In order to support players and stones, a few new classes and tests for future game rules are added. On the new map the test classes are positioned in the north-eastern quadrant, opposite to the other classes. This indicates that the newly added test classes implement

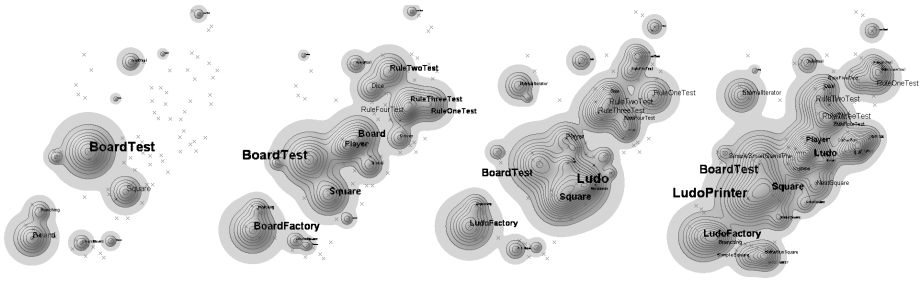


Figure 6.3: From left to right: each map shows four iterations of the same software system. As all four views use the same layout, a user can build up a mental model of the system's spatial structure. For example, `Board/LudoFactory` is on all four views located in the south-western quadrant. See also Figure 5 and 6 for more views of this system.

a novel feature (that is testing of the game's "business rules") and are thus not related to the factory's domain of board initialization.

- During the third iteration (Figure 6.3, second to the right) the actual game rules are implemented. Most rules are implemented in the `Square` and `Ludo` class, thus their mountain rises. In the south-west, we can notice that, although the `BoardFactory` has been renamed to `LudoFactory`, its position on the map has not changed considerably.

- The fourth map (Figure 6.3, rightmost) shows the last iteration. A user interface and a printer class have been added. Since both of them depend on most previous parts of the application they are located in the middle of the map. Since the new UI classes use vocabulary from all parts of the system, the islands are joined into a continent.

The layout of elements remains stable over all four iterations. For example, `Board/LudoFactory` is on all four views located in the south-western quadrant.

6.5 Discussion

Software Cartography, as presented in this chapter, is a spatial representation of software. Our approach visualizes software entities using a consistent layout. Software maps present the entire program and are continuous. Software maps contain visual landmarks that allow developers to find parts of the system perceptually rather then relying on conceptual cues, for example names. Since all software maps of a system use the same layout, maps with thematic overlays can be compared to each other.

The layout of software maps is based on the lexical similarity of software entities. Our algorithm uses a vector-space model to position software entities in an multi-dimensional space, and a combination of the Isomap algorithm [179]

and Multidimensional Scaling [33] to map these positions on a two-dimensional display. Software maps can be generated to depict evolution of a software system over time. We evaluated the visual stability of iteratively generated maps considering four open source case studies.

In spite of the aesthetic appeal of hill shading and contour lines, the main contribution of this chapter is not the cartographic look of software maps. The main contribution of Software Cartography is (i) that cartographic position reflects topical distance of software entities, and (ii) that consistent layout allows different software maps to be easily compared. In this way, software maps reflect world maps in an atlas that exploit the same consistent layout to depict various kinds of thematic information about geographical sites.

Software maps at present are largely static. In a more interactive environment the user can "zoom and pan" through the landscape to see features in closer detail, or navigate to other views of the software. The CodeCanvas prototype by Deline and Rowan offers this feature [53]. All source code is put on a large zoomable canvas and developers zoom and pan around on this canvas as they work on the system. Depending on the zoom level of the canvas a different semantic view of the system is shown. For example, as the canvas is zoomed out the source code fades off and labels with colored icons appear instead to mark the signature of methods.

Selectively displaying features would make the environment more attractive for navigation. Instead of generating all the labels and thematic widgets upfront, users can annotate the map, adding comments and waymarks as they perform their tasks. The CodeBubbles prototype by Bragdon and Reiss offers this feature [34]. Instead of offering one global representation, a custom spatial layout of separate method bodies is unfolded as the developer works on the code. For each working context, that is task, a new view is generated. They offer specialized views for debugging contexts, including a differential debugging mode. They are evaluating their prototype in an ongoing series of user studies, on the first of which is being reported in their current publications [35, 34].

Orientation and layout are presently consistent for a single project only. The usefulness of conventions for establishing consistent layout and orientation (that is "testing" is North-East) that will work across multiple projects, possibly within a reasonably well-defined domain, is an open issue. However, given the findings of the user study presented in the next chapter, we consider it to be more worthwhile to invest in customizable layouts rather than predefined global layouts.

Chapter 7

User Study on Software Cartography

Software visualization can be of great use for understanding and exploring a software system in an intuitive manner. Spatial representation of software is a promising approach of increasing interest. However, little is known about how developers interact with spatial visualizations that are embedded in the IDE. In this chapter, we present a pilot study that explores the use of Software Cartography for program comprehension of an unknown system. We investigated whether developers establish a spatial memory of the system, whether clustering by topic offers a sound base layout, and how developers interact with maps. We report our results in the form of observations, hypotheses, and implications. Key findings are a) that developers made good use of the map to inspect search results and call graphs, and b) that developers found the base layout surprising and often confusing. We conclude with concrete advice for the design of embedded software maps.

In the past decade the software visualization community has developed a rich wealth of visualization approaches [57] and provided evidence of their usefulness for expert tasks, such as reverse engineering, release management or dynamic analysis (for example [178, 48, 161, 151]). Typically, these visualization approaches had been implemented in interactive tools [168]. However most of these tools are stand-alone prototypes that have never been integrated in an IDE (integrated development environment). Little is thus known about the benefits of software visualization for the "end users" in software engineering, that is for everyday programmers. What is lacking is how these techniques support the day to day activities of software developers [176].

In this chapter, we report on a pilot study of a spatial software visualization that is embedded in the IDE. The spatial visualization is based on the Software Cartography approach that has been presented and introduced in previous work [116, 111, 66]. Spatial representation of software is a promising research field of increasing interest [192, 53, 34, **?**, 139, 149], however the respective tools are

either not tightly integrated in an IDE or have not yet been evaluated in a user study. Spatial representation of software is supposed to support developers in establishing a long term, spatial memory of the software system. Developers may use spatial memory to recall the location of software artifacts, and to put thematic map overlays in relation with each other [111].

The scenario of our user study is first contact with an unknown closed-source system. Our main question was whether and how developers make use of the embedded visualization and if our initial assumptions made when designing the visualization (as for example the choice of lexical similarity as the map's base layout, see Section 6.3) are based on a valid model of developer needs. Participants had 90 minutes to solve 5 exploratory tasks and to fix one bug report. We used the think-aloud protocol and recorded the voices of the participants together with a screen capture of their IDE interactions. We took manual notes of IDE interaction sequences and annotated the sequences with the recorded think-aloud transcripts.

Results are mixed — some support and some challenge our assumptions on how developers would use the embedded visualization. Participants found the map most useful to explore search results and call graphs, but only rarely used the map for direct navigation contrary to our expectations.

7.1 The Codemap Prototype

Software Cartography uses a spatial visualization of software systems to provide software development teams with a stable and shared mental model. The basic idea of cartographic visualization is to apply thematic cartography [172] to software visualization. That is, to show thematic overlays on top of a stable, spatial base layout. Features on a thematic map are either point-based, arrow-based or continuous. For software this could be the dispersion of design flaws as visualized using icons; a call graph is visualized as a flow map (as illustrated on Figure 7.1); and test coverage is visualized as a choropleth map, that is a heat map.

Software Cartography is most useful when it supports as many development tasks with spatial location awareness as possible. We therefore integrated our prototype into the Eclipse IDE so that a map of the software system may always be present. This helps developers to correlate as many development tasks as possible with their spatial location.

At the moment, the Codemap plug-in for Eclipse supports the following tasks:[1]

- Navigation within a software system, be it for development or analysis. Codemap is integrated with the package explorer and editor of Eclipse. The selection in the package explorer and the selection on the map are linked. Open files are marked with an icon on the map. Double clicking

[1] http://scg.unibe.ch/codemap

on the map opens the closest file in the editor. When using heat map mode, recently visited classes are highlighted on the map.

- Comparing software metrics to each other, for example to compare bug density with code coverage. The map displays search results, compiler errors, and (given the Eclemma plug-in is installed) test coverage information. More information can be added through a plug-in extension point.

- Social awareness of collaboration in the development team. CODEMAP can connect two or more Eclipse instances to show open files of other developers. Colored icons are used to show the currently open files of all developers. The icons are updated in real time.

- Understand a software system's domain. The layout of CODEMAP is based on clustering software by topic [110], as it has been shown that, over time, the lexicon of source code is more stable than its structure [8]. Labels on the map are not limited to class names, but include automatically retrieved keywords and topics.

- Exploring a system during reverse engineering. CODEMAP is integrated with Eclipse's structural navigation features, such as search for callers, implementers, and references. Arrows are shown for search results. We apply the FLOW MAP algorithm [154] to avoid visual clutter by merging parallel arrow edges. Figure 7.1 shows the result of searching for calls to the `#getSettingOrDefault` method in the `MenuAction` class.

7.2 Methodology

We evaluated our approach in a pilot study with professional developers and students. The scenario investigated by the experiment is first contact with an unknown software system. Participants have 90 minutes to solve 5 program comprehension tasks and to fix one bug report. After the experiment, participants are asked to sketch their mental map of the system.

Our goal for the present pilot study was to learn about the usability of CODEMAP for program comprehension. We have been seeking to answer several questions. How can we support developers in establishing a spatial memory of software systems? How do we best support the developers spatial memory using software visualization? How to best embed spatial software visualization in the IDE? When provided with spatial representation of search results and call graphs, how do developers make use of them?

Not covered in this study, and thus open for future user studies, are the shared team awareness and long term memory claims of the Software Cartography approach [111].

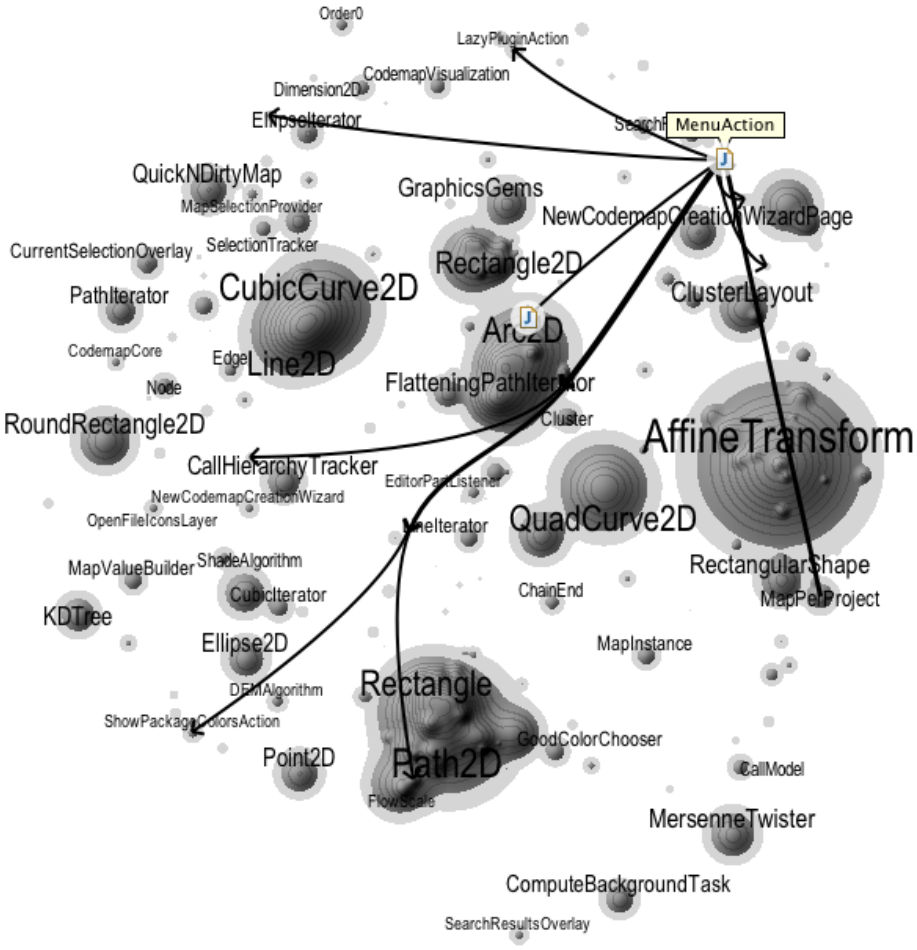


Figure 7.1: *Thematic codemap of a software system. Here the* Codemap *tool itself is shown. Arrow edges show incoming calls to the* `#getSettingOrDefault` *method in the* `MenuAction` *class, which is currently active in the editor and thus labeled with a tool-tip.*

7.2.1 Design of the Study

The study consists of six programming tasks. The training task introduced the participants to the CODEMAP plug-in. The first five tasks were program comprehension tasks, starting with general questions and then going into more and more detailed questions. Eventually, the last task was to fix an actual bug in the system. Participants were asked to use the map whenever they saw fit, but otherwise they were free to use any other feature of Eclipse they wanted.

Task 1, Domain and Collaborators *"Find the purpose of the given application and identify the main collaborators. Explore the system, determine its domain, and fulfil the following tasks: a) describe the domain, b) list the main collaborators, c) draw a simple collaboration diagram, d) identify the main feature of the application."*

Task 2, Technologies *"In this task we are interested in the technologies used in the application. List the main technologies, such as for example Ajax, XML, or unit testing."*

Task 3, Architecture *"In this task we are going to take a look at the architecture of the application. Reverse engineer the architecture by answering the following questions: a) which architectural paradigm is used (as for example pipes and filters, layers, big ball of mud, etc)? b) what are the main architectural components? c) how are those components related to one another? d) draw a UML diagram at the level of components."*

Task 4, Feature Location *"In this task we are interested in classes that collaborate in a given feature. Please locate the following features: a) Interactive users are reminded after some months, and eventually deleted if they do not log in after a certain number of months, b) Depending on the kind of user, a user can see and edit more or less data. There are permission settings for each kind of user that are checked whenever data is accesses, and c) Active search: the system compares the curriculum vitae of the users with stored searches of the companies and mails new matches to the companies."*

Task 5, Code Assessment *"In this task we want to assess the code quality of the application. Please answer the following questions: a) what is the degree of test coverage? b) are there any god classes? c) are the classes organized in their proper packages? Should certain classes be moved to other packages? Please list two to three examples."*

We provided a code coverage plug-in with the experiment, as well as a definition of what constitutes a god class [121].

Task 6, Bug Fixing In this task we provided an actual bug report and asked *"Describe how you would handle the bug report, that is how and where you would*

change the system and which classes are involved in the bug fix. You are not asked to actually fix the bug, but just to describe how you would fix it."

7.2.2 Participant Selection

Participants were selected through an open call for participation on Twitter[2] as well as through flyers distributed at a local Eclipse event. Subjects were required to be medium level Java programmers with at least one year of experience with both Java and Eclipse programming. The six tasks had been designed so that the participants did not need to be knowledgeable with the provided application, but rather that they explore it as they go along. Seven participants took part in the experiment: 4 graduate students and 3 professional developers from industry. None of the participants was familiar with the provided application or with the Codemap plugin; even though some had attended a 15 minute presentation about the Codemap plugin at the Eclipse event mentioned above.

7.2.3 Study Setting

The study consisted of three main parts. The first part was the training task in which the participants were given a short presentation of Codemap and a tutorial document that explained all features of the Codemap plug-in. The tutorial explained all features mentioned in Section 7.1 using walk-through descriptions of their use. The participants were given 20 minutes to explore a small example program using the Codemap plug-in. When they felt ready, we started part two of the experiment.

The second part consisted of the actual programming tasks. A fixed amount of time was allotted to each task. Participants were asked to spend no more than 15 minutes on each task. All subjects had access to the Codemap plugin as our aim was to explore their use of the plugin rather than to compare a controlled parameter against the baseline.

Eventually, in a third part we held a debriefing session. We asked participants to draw a map (with any layout or diagram language whatsoever) of how they would explain the system under study to another developer. We asked the participants for feedback regarding their use of the Codemap plugin and how the plugin could be improved.

7.3 Data Collection

We asked the participants to think aloud, and recorded their voice together with a captured video of their computer screen using the Camtasia software[3]. We reminded the participants to think aloud whenever they fell silent: we told them to imagine a junior programmer sitting beside them to whom they are to explain their actions (Master/Apprentice [27]). The participants were asked to respond

[2] http://twitter.com/codemap
[3] http://www.techsmith.com/camtasia

to a survey while performing the study. The survey consisted of their answers to the tasks, as well as the perceived difficulty of the tasks and whether they found the Codemap plugin useful for the task at hand. We used a combination of semantic differential statements and Likert scales with a 5 point scale.

We measured whether or not subjects were successful in completing a programming task. We used three success levels to measure the success and failure of tasks: a task could be a success, a partial success or a failure. We further subdivided tasks 4 and 5 into three subtasks and recorded success levels for each individual subtask. We asked one of the original authors of the system to assess the success levels. As this was a think-aloud study, we did not measure time, but alloted a fixed 15 minute slot to each task.

Our main interest was focused on how the participants used the IDE to solve the tasks, independent of their success level. To do this, we transcribed important quotes from the recorded participant voices and screen captures and took notes of the actions that the participants did during the tasks. For each task we tracked the use of the following IDE elements:

- Browsing the system using the *Package Explorer* and *Outline* view. This includes both drill-down as well as linear browsing of package, class and method names.

- Browsing the system using the spatial visualization of the Codemap plugin. This includes both opening single classes, selecting a whole cluster of classes on the map, as well as reading class name labels on the map.

- Reading source code in the editor pane, including documentation in the comments of class and method headers.

- Navigating the structure of the system using the *Type Hierarchy* and *Call Hierarchy* view. We tracked whether they explored the results of these searches in Eclipse's tabular result view or using the flow-map arrows displayed on the spatial visualization of Codemap.

- Searching the structure of the system with either the *Open Type* or *Java Search* dialog. This allows users to search for specific structural elements such as classes, methods or fields. Again, we tracked whether they explored the results in Eclipse's result view or on the visualization of Codemap.

- Searching the system with the unstructured text search, either through the *Java Search* dialog or the immediate search bar of the Codemap plugin. Also here, we tracked whether they explored the results in Eclipse's result view or on the visualization of Codemap.

Replicability: the raw data of our analysis is available on the CODEMAP website at http://scg.unibe.ch/codemap.

7.4 Results

After analyzing our data, we observed different degrees of interaction with the CODEMAP plug-in. We focused our analysis on interaction sequences that included interaction with the CODEMAP plug-in, but also on those interaction sequences that challenged our assumptions about how developers would make use of the plug-in.

The presentation of results is structured as follows. First, we briefly cover how each task was solved. Then present an in-depth analysis of our observations, structured by triples of *observation*, *hypothesis*, and *implication*. Implications are directed at improving the design and usability of spatial visualizations that are embedded in an IDE.

7.4.1 Task Performance

Task 1, Domain and Collaborators Participants used an approach best described as a "reverse Booch method" [32]. Given a two-sentence description of the system that we've provided, they searched for nouns and verbs using Eclipse's full text search. Most participants used CODEMAP to assess quantity and dispersion of search results, and also to directly select and inspect large classes. Then they looked at the class names of the matches to learn about the domain and collaborators of the system. Students also read source code, whereas professional participants limited their investigation to using the package explorer and class outline.

Task 2, Technologies This task showed the most uniform behavior from both student and professional participants. They inspected the build path node and opened all included JAR libraries. Professional developers typically raised the concern that possibly not all of these libraries were (still) used and started to explore whether they were used. Typically they would carry out a search to do so, but one developer showed a very interesting pattern: He would remove the libary "on purpose" and then look for compile errors as an indicator of its use. Students seems to implicitly assume that all libraries were actually used, at least they never raised such a concern. We interpret this as a sign that professionals are more cautious [105] and thus more aware of the typical decay caused by software evolution, which may include dead libraries.

Task 3, Architecture Typically participants drilled-down with the package explorer and read all package names. All professionals started out by formulating the hypothesis of a layered three-tier architecture, and then start fitting the packages to the different layers. Most participants used CODEMAP to look at the dispersion of a package's classes (when selecting a package in the package explorer, the contained classes are highlighted on the map).

To learn about the architectural constraints, professionals, for the first time in the experiment, started reading source code. They also did so quite differently from the way that students did. Whereas students typically read code line by

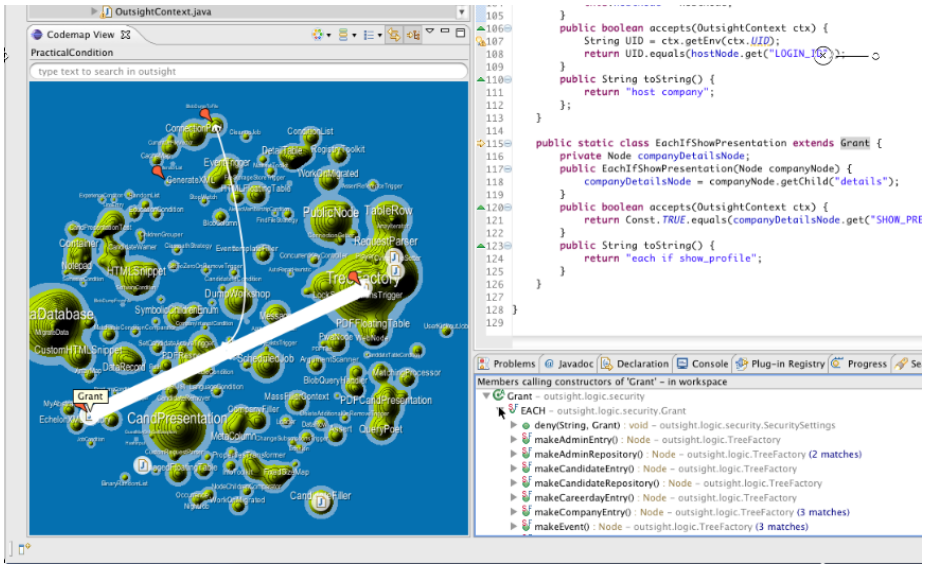


Figure 7.2: *Screen capture of "Aha moment" as encountered by participant T during task 4-b (location of security features): Upon opening the call hierarchy of* `Grant`*'s constructor, a huge call-arrow appeared on the map: indicating dozens of individual calls that connect the security-related archipelago in the south-west with the* `TreeFactory` *island in the east. Given the visual evidence of this arrow, participant T solved the task without further investigation.*

line, trying to understand what it does, the professionals rather used the scroll-wheel to skim over the code as it flies by on the screen, thereby looking for "landmarks" such as constructor calls, method signatures and field definitions. Professionals made much more use of "open call hierarchy" and "open type hierarchy". Interestingly enough, only one participant opened a type hierarchy of the whole project.

Task 4, Feature Location For this task, participants made most frequent and more interesting use of CODEMAP than for any other task. As in task 1, participants used a reversal of the Booch method. They searched for nouns and verbs found in the feature description. Again, they used the map to assess quantity and dispersion of search results. Also two participants used the map to select and inspect search matches based on their context in the map.

Participants now began to read more source code than before. In particular, when they found a promising search result they used the "open call hierarchy" feature to locate related classes. All participants reported that CODEMAP flow-map overlay helped them to work with the call graph. For some developers there was an actual "Aha moment" where one glance at the CODEMAP helped

them to solve the current subtask immediately without further investigation. Figure 7.2 illustrates one particular moment as encountered by participant T during the location of the security feature.

Task 5, Code Assesment This set of tasks made it most obvious that CODEMAP's layout was not based on package structure. Participants reported that they had a hard time to interpret the thematic maps as they could not map locations on the map to packages. In particular the professional participants expressed concerns regarding the use of KLOC for hill size. They expressed concerns that this might be misleading since lines of code is not always an indicator of importance or centrality on the system's design.

Task 6, Bug Fixing Participants mainly used the same approach as for the feature location tasks. They first located the implementation of the feature in which the bug occurs, and then fixed the bug. Professional participants did so successfully, whereas student participants did not manage to find the correct location in the source code.

Wrap-up session 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 cue of quantity and distribution of your search results"*. In fact, we observed that that participants rarely used the map for direct navigation but often for search and reverse engineering tasks.

Another observation was that inexperienced developers (that is students) are more likely to find the map useful than professional developers. This might be explained by the hypothesis that to power users *any* new way of using the IDE is likely to slow them down, and conversely to 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 search, that was appreciated and used by all participants but one who preferred to use the search dialog.

One participant also provided us feedback comparing his experience with CODEMAP to that with the Moose analysis tool [148]. He uses Moose at work after having attended a tutorial by a consultant. He said he prefers the immediate feedback of Codemap, and reported that *"the gap between Moose and IDE is just too large, not to mention the struggle of importing Java code. Moose helps you to for example find god-classes but this is typically not new to developers that know a system. Codemap seems more interesting as it integrates with what you actually do in the IDE as you program."*

7.4.2 Observations, Hypotheses, Implications

In this section, we present an in-depth analysis of our observations, structured by triples of *observation*, *hypothesis*, and *implication*. Implications are directed

at improving the design and usability of spatial visualizations that are embedded in an IDE.

Observation 6.1: When thinking aloud, developers spoke of the system's architecture in spatial terms The think-aloud protocol revealed that participants refer to the system's architecture in spatial terms. Professional participants referred to packages as being above, below, or at the some level as one another. Some of them even did so before recovering the system's 3-tier architecture in task #3. Most professionals referred to utility packages a being spatially beside or outside the layered architecture.

For example, participant T located all utility packages in the upper left corner, separated by a jagged line. While doing so, he made a gesture as if pushing the utility classes away and stated, *"I am putting them up here because to me they are somehow beside the system."*

Students on the other hand made much fewer references to the system's architecture, both spatial as well as in general. They were typically reasoning about the system at the level of classes and source lines, rather than in architectural terms. The maps drawn by students in the wrap-up phase, however, showed similar spatial structure to those of the professionals. It remains thus open whether students established a genuine spatial model while working with the code (as we observed for professionals) or only because they were asked to draw the wrap-up maps.

Hypothesis 6.1: Professional developers establish a spatial mental model of the system's architecture Based on above observations there is evidence to assume that professional developers establish a spatial mental model of the system's architecture as they work with code. Furthermore, they do so even without visual aids, since they use spatial terms and thinking even before being asked to draw a diagram of the system's architecture.

Implication 6.1: Developers should be able to arrange the layout according to their mental model This has implications on the design of a system's spatial visualization. Developers should be able to arrange the layout according to their mental model. Developers should be able to drag and move parts of the map around as they wish, rather than having to stick with the automatically established layout. Code Canvas [53] and Code Bubbles [34] both already address this implication. In those tools, the user may drag individual elements around and arrange them according to his mental model.

We observed that developers referred to architectural components, but not classes, in spatial terms. The needs of developers might thus be even better served by providing them more high-level means of arranging the map. Our next prototype will use *anchored multidimensional scaling* such that developers may initialize the map to their mental model. Anchored MDS allows the developer to define anchors which influence the layout of the map [38, Sec 4.4]. Any software artifact can be used as an anchor (as long as we can compute a its

distance to artifacts on the map), even for example external libraries. In this way, developers might for example arrange the database layer in the south and the UI layer in the north using the respective libraries as anchors.

Observation 6.2: Participants used Codemap to assess quantity and dispersion of search results and call graphs The feature of Codemap that was used most often, by both professionals and students, was the illustration of search results and call graphs. Participants reported that they liked the search-result support of the map, explaining that it gives them much faster initial feedback than Eclipse's tabular presentation of search results. Many participants reported that it was *"as if you could feel the search results,"* and that *"you get an immediate estimate how much was found, whether it is all one place or scattered all over the place."*

Figure 7.2 illustrates one particular "Aha moment" as encountered by participant T during task 4-b, that is location of security features: Upon opening the call hierarchy, a huge call-arrow appeared on the map: indicating dozens of individual calls that connect the security-related archipelago in the south-west with the `TreeFactory` island in the east. Given the visual evidence of this arrow, the participant solved the task immediately without further investigation of the system.

Hypothesis 6.2: Intuitive visualizations to show quantity and dispersion of search results (as well as call graphs) address an important need of developers Given the above observation it seems clear that developers have urgent needs for better representation of search results than tabular lists. We found that both students and professionals used the map to get an immediate estimation of search results. This is most interesting since otherwise their use of the tabular search results differed: Professionals glanced at the results, inspected one or maybe two results, and then either accepted or rejected their hypothesis about the system, while students would resort to a linear search through all search results, not daring to reject a hypothesis on the grounds of one or two inspected results only.

Given the map's illustration of search results however, the behavior of both groups changed. Students dared to take quick decisions from a mere glance at the map, whereas professionals were more likely to inspect several results. One professional reported that he *"inspected more results than usual, because the map shows them in their context and that this helps him to take a more informed choice on which results are worth inspection and which ones not."*

Implication 6.2: Tools should put search results into a meaningful context, so developers can take both quicker and better-informed decisions The need for better presentation of search results has implications beyond the design of spatial visualizations. Work on presentation of search results goes beyond spatial maps [84], for example results can be presented as

a graph. Poshyvanyk and Marcus [158] have taken one such approach (representing search results as a lattice) and applied it to source code search with promising results.

For our next prototype we plan to integrate search results into the package explorer view, just as is already done with compile errors (which are, from this point of view, just like the search results of a complex query that is run to find syntax errors). This planned feature addresses another implication of our study as well, as we have found that some developers establish a spatial memory of the package explorer view. It therefore makes sense to mark search results both on our map as well as in the explorer view.

Observation 6.3/a: When interacting with the map, participants were attracted to isolated elements, rather than exploring clusters of closely related elements We found that participants are more likely to inspect easily discernible elements on the map. They are more likely to notice and interact with an isolated island rather than with elements that are part of a larger continent. Unfortunately, it is exactly dense and strongly correlated clusters that contain the most interesting parts of the system! When investigating this issue, participants answered that *"those (isolated) elements looked more important as they visually stick out of the rest of the map."*

Also, when working with another system that had (unlike the present study) a large cluster in the middle surrounded by archipelagos on the periphery, we found that users started their exploration with isolated hills in the periphery, only then working their way towards the more dense cluster in the middle.

Hypothesis 6.3/b: Participants rarely used Codemap to return to previously visited locations, instead using package explorer and "Open Type" to do so Contrary to our assumptions, participants did not use the map to return to previously visited locations by recalling them from spatial memory. Some would use the map, but only for exposed classes that are easily recognizable and clickable. This observation is related to the previous one.

We found however some participants established a spatial memory of the package explorer — and did so *in addition to their spatial model of the system's architecture!* For example, participant S would drill down with the explorer saying "let's open that class down there" or "there was this class up here." Over the course of the experiment he got quicker at navigating back to previously visited classes in the package explorer. Other participants, as for example participant T, relied on lexical cues and made extensive use of Eclipse's "Open Type" dialog to find their way back to previously visited classes.

Usability glitches will of course worsen the effect of (or might even be the main cause of) not using the map for navigation and revisiting classes. From this it follows that:

Hypothesis 6.3: Developers avoided clusters of closely related elements because they are difficult to identify and select on the map

All participants had difficulties to open files by clicking on the map. They had difficulties to select classes on the map when they are in a crowded cluster. They would click in the middle of a label, but often the labels are not centered, which is an unavoidable artifact of any labeling algorithm, and thus the clicks would open a different (unlabeled) class.

Codemap does provide tooltips, however participants did not use them. From observing their work it was obvious why: both students and professionals were working at such a speed that waiting for a tooltip to appear would have totally taken them out of their workflow.

Implication 6.3: The map's layout should be such that all elements are easily discernable and easy to click Real estate on a computer screen is limited, and even more so in an IDE with all its views and panels. As tool builders we have limited space available for an embedded visualization. Given our goal of establishing a global layout we face the challenge of having to visualize all elements of a system in that very limited space.

The current implementation of Codemap has one level of scale only, which may yield crowded clusters where elements are placed just pixels apart. A zoomable map as provided by Code Canvas [53] addresses this issue.

The fact that we are attracted by elements that are visually detached has two impacts: one is that we tend to look at isolated elements as being of low significance, the other being that it is hard to identify elements in the cluster. These impacts are very different, but can both be addressed in a common way. For instance, a threshold could be used to not show isolated elements at all, but only significant clusters. Alternatively, colors may be used to display isolated elements so that they do not draw our attention so readily.

Observation 6.4: Participants rarely used Codemap to return to previously visited locations, instead using package explorer and "Open Type" to do so

Hypothesis 6.4: Developers failed to establish a spatial memory of the map not due to its layout but also due to missing visual cues In general, the issue of usability is orthogonal to the map's layout. For example, offering "search as you type" might help to raise map interaction for those developers that mainly rely on lexical cues, no matter which base layout is used. It was our impression that any exploratory user study of embedded software visualization will be dominated just as by usability issues as by technical factors, such as your choice of layout algorithm.

Implication 6.4: It should be possible to bookmark classes as "top landmarks," both manually and automatically based on usage statistics What might help as well to ease the retrieval of previously visited classes is a "top landmarks" feature where you can manually (but also automatically

based on visits) set markers on the map as starting points for further activities. We plan to work on this for our next prototype.

Observation 6.5: Participants used Codemap as if its layout were based on package structure — even though they were aware of the underlying topic-based layout Developers assume that packages are a valid decomposition of the system and expect that the layout of the spatial visualization corresponds to the package structure. We found that clustering classes by topic rather than packages violates the "principle of least surprise." We observed that participants tended to interpret visual distance as a measure of structural dependencies — even though they were aware of the underlying lexical implementation!

Participants expected the layout to reflect at least some structural property. Most of them reacted surprised or confused when for example the classes of a package were not mostly in the same place. For example, Participant S reported in the wrap-up, *"this is a very useful tool but the layout does not make sense".* Another participant stated during task 3 (that is the architecture recovery) with confusion that *"the classes contained in packages are scattered on the map, it is not obvious what their spatial connection is."*

Hypothesis 6.5: From the developers view, the predominant mental decomposition of a system is package structure Given the general stance of reverse engineering research [148, 110, 61] we had come to distrust package decomposition, however it seems that developers like to rely on the packaging that other developers have made when designing the system.

One problem raised by research in re-packaging legacy systems is that packages play too many roles: as distribution units, as units of namespacing, as working sets, as topics, as unit of architectural components, etc. However, as an opposing point of view, we can relate packaging to the folksonomies of the Web 2.0, where users label elements with unstructured tags that are then exploited by other users to search for elements. In the same way, we could say that putting trust into a given package structure is a way of collaborative filtering. Developers assume that other developers had made the same choice as they would when packaging the system.

Implication 6.5: The map layout should be based on code structure rather than latent topics only. However, non-structural data should be used to enrich the layout When running the user study, it became quickly apparent that we should revise our initial assumption that lexical similarity is a valid dissimilarity metric for the spatial layout. This was the strongest feedback, and as is often the case in exploratory user studies, already obvious from watching the first professional participant for five minutes only. From all participants we got the feedback that they expect the layout to be structural and that our clustering by topics kept surprising them even after working with the map for almost two hours.

Still we think that spatial layouts that go beyond package structure are worthwhile. Therefore, we propose to enrich structure-based layout with non-structural data, such as design flaws. For future work, we are about to refine our layout algorithm based on that conclusion. The new layout is based on both lexical similarity and the ideal structural proximity proposed by the "Law of Demeter" (LOD). This is a design guideline that states that each method should only talk to its friends, which are defined as its class's fields, its local variables and its method arguments. Based on this we can defined an *idealized* call-based distance between software artifacts. Given a LOD-based layout, software artifacts are close to one another if they are supposed to call one another and far apart if they better should not call one another. Thus we get the desired property that visualizing call-graphs conveys meaningful arrow distances. On a LOD-based map, any long-distance-call has a diagnostic interpretation that helps developers to take actions: Long flow-map arrows indicate calls that possibly violate the "Law of Demeter".

7.5 Threats to Validity

This section summarizes threats to validity. The study had a small sample size (3 students, 4 professionals) and might thus not be representative. We manually evaluated the data, results might thus be biased. Nevertheless, results are promising and running a pilot think-aloud study with a small user group is a state-of-the-art technique in usability engineering to learn learn about the reactions of users. Such pilot studies are typically used as feedback for further iteration of the tool and to assess the usefulness of its application [147].

7.6 Conclusion

In this paper we presented an evaluation of spatial software visualization in the IDE. We embedded a prototype of the *Software Cartography* approach [116, 66, 111], the CODEMAP plug-in, in the Eclipse IDE and ran an exploratory user study which included both students and professionals.

Software maps are supposed to help developers with a visual representation of their software systems that addresses their spatial thinking and memory. The scenario of our user study was first contact with an unknown closed-source system. Results were as follows:

- Participants made good use of the map to inspect search results and call graph. They reported that the spatial visualization provided them with an immediate estimate of quantity and dispersion of search results.
- Participants found the layout of the map (which uses lexical information to cluster classes by topic) surprising and often confusing. This led to the revision of our initial assumption that lexical similarity is sufficient to lay out the cartographic map.

We drew the following four main observations, and concluded from these the following implications:

- All participants used a form of spacial thinking to understand the system. It would be best to allow developers to rearrange the initial layout according to their spatial memory.
- Immediate estimate of quantity and dispersion of search results is useful and the map suits this well.
- Participants are distracted by isolated elements, which do not appear in textual/tabular representation. This is the drawback of visualization, which must find the right balance between the power of visualization and the pitfall of visualization. The map should be improved to mitigate that.
- The coexistence of two models for the software (one structural, one conceptual) causes some confusion. With the present map and implementation, participants were puzzled by non-structural nature of the map.

Developers intuitively expect that the map meets their mental model of the system's architecture. We observed that if this is not given, developers are not able to take advantage of the map's consistent layout. So for example, even though north/south and east/west directions had clear (semantic) interpretations in the map used for the user study, developers did not navigate along these axes.

However, even with the most perfect layout, developers might not be able to take advantage of the map if the elements are barely discernable, and thus difficult to inspect.

Based on the results of our user study, we conclude with the working hypothesis that *the map should incorporate structural information and be improved from point of usability* and that *we need more work to make two models (one structural, one conceptual) co-exists without creating confusion.*

In order to achieve this, we propose the following changes to the Software Cartography approach:

- Compute the initial layout so that distance reflects structural correlation, since this is what the developers expect (that is "principle of least astonishment").
- Use *anchored multi-dimensional scaling* for the layout so that developers may rearrange the map according to their spatial model of the system.
- Use architectural components as anchors for rearrangement of the map, since spatial thinking of developers is strongest at the architectural level.
- Improve the usability experience of inspecting and selecting of elements on the map, possibly using a zoomable user interface.

Chapter 8

Stories of Collaboration

Aim	Learning about team collaboration.
Reach	Team of a project's local codebase.
Clues	Episodic (established through social and historical information).
Query	Visual analytics of a story-telling visualization.

Episodic clues are of great help to developers when finding their way through a system. However episodic memory is only available to those developers who know a system's history first-hand. For most software systems, the tribal folklore of the team is the only record on the system's history beside the version control system. The tribal folklore is hard to query because it is only present in the team's minds. The version control system is hard to query because it is typically only present as a series of textual low-level changes to the system's source code. There is a need for better means of recovering and telling a system's story to both new hires and seasoned team members.

In this chapter we present an approach to visualize a system history and its team's collaboration as a story-telling visualization. Story-telling visualizations are a branch of information visualization that has been popularized by the information graphics of newspapers such as the New York Times and the Guardian. A story-telling visualization is supposed to invite its reader to get engaged with the visualized data by establishing a personal connection between the reader and the presented data. Our approach uses social and historical information taken from the version control system to establish an episodic visualization of the system's history. We show the lifeline of files ordered by code ownership, thus telling the story of the team's collaboration. Teams are typically getting very engaged and excited when shown the visualization of their own system. Both new hires and seasoned team members can use this visualization to learn about episodes from the system's history in order to take better technical decisions when working with the system in the future.

As systems evolve their structure changes in ways not expected upfront. As time goes by, the knowledge of the developers becomes more and more critical for

the process of understanding the system. That is, when we want to understand a certain issue of the system we ask the knowledgeable developers. Yet, in large systems, not every developer is knowledgeable in all the details of the system. Thus, we would want to know which developer is knowledgeable in the issue at hand. In this chapter we make use of the mapping between the changes and the author identifiers (for example user names) provided by versioning repositories. We first define a measurement for the notion of code ownership. We use this measurement to define the *Ownership Map* visualization to understand when and how different developers interacted in which way and in which part of the system[1]. We report the results we obtained for several large systems.

Software systems need to change in ways that challenge the original design. Even if the original documentation exists, it might not reflect the code anymore. In such situations, it is crucial to get access to developer knowledge to understand the system. As systems grow larger, not all developers know about the entire system, thus, to make the best use of developer knowledge, we need to know which developer is knowledgeable in which part of the system.

From another perspective, Conway's law [45] states that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." That is, the shape of the organization reflects on the shape of the system. As such, to understand the system, one also has to understand the interaction between the developers and the system [55].

In this chapter we aim to understand how the developers drove the evolution of the system. In particular we provide answers to the following questions:

- How many authors developed the system?
- Which author developed which part of the system?
- What were the interactions between the developers?

In our approach, we assume that the original developer of a line of code is the most knowledgeable in that line of code. We use this assumption to determine the owner of a piece of code (for example a file) as being the developer that owns the largest part of that piece of code. We make use of the ownership to provide a visualization that helps to understand how developers interacted with the system. The visualization represents files as lines, and colors these lines according to the ownership over time.

Contrary to similar approaches [185], we give a semantic order to the file axis (that is we do not rely on the names of the files) by clustering the files based on their history of changes: files committed in the same period are related [72].

We implemented our approach in Chronia, a tool built on top of the Moose reengineering environment [59]. Our implementation relies on the CVS version control system. Our aim was to provide a solution that gives fast results, therefore, our approach relies only on information from the CVS log without checking

[1]The visualizations in this chapter make heavy use of colors. Please obtain a color-printed or electronic version for better understanding.

out the whole repository. As a consequence, we can analyze large systems in a very short period of time, making the approach usable in the early stages of reverse engineering.

To show the usefulness of our solution we applied it for several large case studies. We report here some of the findings and discuss different facets of the approach.

The contributions of the chapter are:

- The definition of file ownership.
- The clustering of files based on their commit history.
- A characterization of developer behaviors.
- The *Ownership Map* visualization.

8.1 Data Extraction from CVS log

This section introduces a measurement to characterize the code ownership. The assumption is that the original developer of a line of code is the most knowledgeable in that line of code. Based on this assumption, we determine the owner of a piece of code as being the developer who owns the most lines of that piece of code.

Our approach is based on incremental line change information taken from the log files of the version control system. We approximate the ownership of files from this information. An advantage of this approach is that we can work with the log files only and do not need to retrieve the entire version history. Other than more modern systems like Git and Subversion, the CVS protocol does not provide an easy way to mirror an entire repository [30].

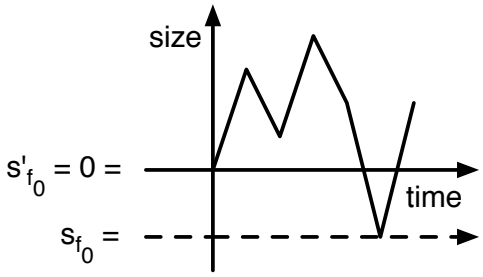
Below we present a snippet from a CVS log. The log lists for each version f_n of a file, termed revision in CVS, the time t_{f_n} of its commit, the name of its author α_{f_n}, some state information and finally the number of added and removed lines as deltas a_{f_n} and r_{f_n}. We use these numbers to recover both the file size s_{f_n}, and the code ownership $own^{\alpha}_{f_n}$.

```
----------------------------
revision 1.38
date: 2005/04/20 13:11:24;  author: girba;  state: Exp;  lines: +36 -11
added implementation section
----------------------------
revision 1.37
date: 2005/04/20 11:45:22;  author: akuhn;  state: Exp;  lines: +4 -5
fixed errors in ownership formula
----------------------------
revision 1.36
date: 2005/04/20 07:49:58;  author: mseeberg;  state: Exp;  lines: +16 -16
Fixed math to get pdflatex through without errors.
----------------------------
```

8.1.1 Measuring File Size

Let s_{f_n} be the size of revision f_n, measured in number of lines. The number of lines is not given in the CVS log, but can be computed from the deltas a_{f_n}

and r_{f_n} of added and removed lines. Even though the CVS log does not give the initial size s_{f_0}, we can give an estimate based on the fact that one cannot remove more lines from a file than were ever contained. We define s_{f_n} as in Figure 8.1: we first calculate the sizes starting with an initial size of 0, and then in a second pass adjust the values with the lowest value encountered in the first pass.



$$s'_{f_0} := 0$$
$$s'_{f_n} := s'_{f_{n-1}} + a_{f_{n-1}} - r_{f_n}$$
$$s_{f_0} := |min\{s'_x\}|$$
$$s_{f_n} := s_{f_{n-1}} + a_{f_n} - r_{f_n}$$

Figure 8.1: The computation of the initial size.

This is a pessimistic estimate, since lines that never changed are not covered by the deltas in the CVS log. This is an acceptable assumption since our main focus is telling the story of the developers, not measuring lines that were never touched by a developer. Furthermore in a long-living system the content of files is entirely replaced or rewritten at least once if not several times. Thus the estimate matches the correct size of most files.

8.1.2 Measuring Code Ownership

A developer owns a line of code if he was the last one who committed a change to that line. In the same way, we define file ownership as the percentage of lines he owns in a file. And the overall owner of a file is the developer who owns the largest part of it.

Let $own^{\delta}_{f_n}$ be the percentage of lines in revision f_n owned by author δ. Given the file size s_{f_n}, and both the author δ_{f_n} who committed the change and a_{f_n} the number of lines he added, we defined ownership as:

$$own^{\delta}_{f_0} := \begin{cases} 1, & \delta = \delta_{f_0} \\ 0, & else \end{cases}$$

$$own^{\delta}_{f_n} := own^{\delta}_{f_{n-1}} \frac{s_{f_n} - a_{f_n}}{s_{f_n}} + \begin{cases} \frac{a_{f_n}}{s_{f_n}}, & \delta = \delta_{f_n} \\ 0, & else \end{cases}$$

In the definition we assume uniform distribution of removed lines r_{f_n} among the preceding owners of f_{n-1}. A better estimate than $own^{\alpha}_{f_n}$ can be retrieved by checking out the content of each revision and using a diff algorithm to find out to whom the removed lines actually belonged. But this natively supported by the CVS protocol and this time consuming.

8.2 The Ownership Map View

We introduce the *Ownership Map* visualization as in Figure 8.2. The visualization is similar to the Evolution Matrix [119]: each line represents a history of a file, and each circle on a line represents a change to that file.

The color of the circle denotes the author who made the change. The size of the circle reflects the proportion of the file that got changed. That is, the larger the change the larger the circle. And the color of the line denotes the author who owns most of the file.

Bertin [25] assessed that one of the good practices in information visualization is to offer to the viewer visualizations that can be grasped at one glance. The colors used in our visualizations follow visual guidelines suggested by Bertin, Tufte [181], and Ware [190] - for example we take into account that the human brain is not capable of processing more than a dozen distinct colors.

In a large system, we can have hundreds of developers. Because the human eye is not capable of distinguishing that many colors, we only display the authors who committed most of all changes using distinct colors; the remaining authors are represented in gray. Furthermore, we also represent those files with gray that came into the CVS repository with the initial import. Imported files are usually sources from another project with unknown authors and are thus not necessarily created by the author that performed the import. In short, a gray line represents either an unknown owner, or an unimportant one.

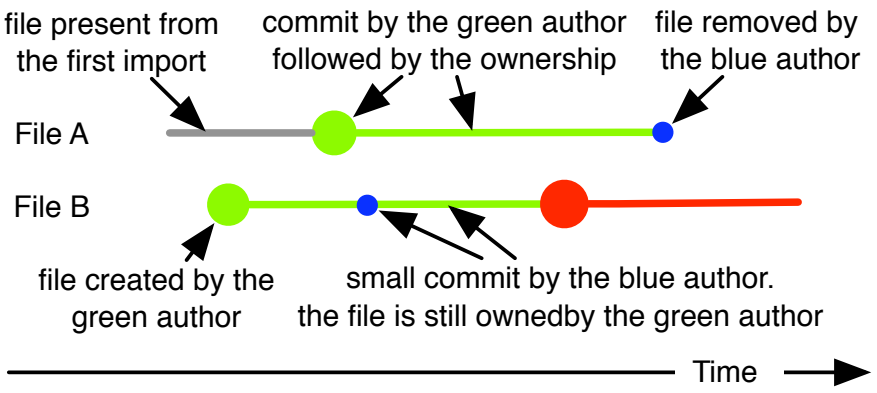


Figure 8.2: Example of ownership visualization of two files.

In the example from Figure 8.2, each line represents the lifetime of a file; each circle represents a change. File A appears gray in the first part as it originates from the initial import. Later the green author significantly changed the file, and he became the owner of the file. In the end, the blue author deleted the file. File B was created by the green author. Afterwards, the blue author changed the file, but still the green author owns the larger part, so the line remains green. At some point, the red author committed a large change and took over

the ownership. The file was not deleted.

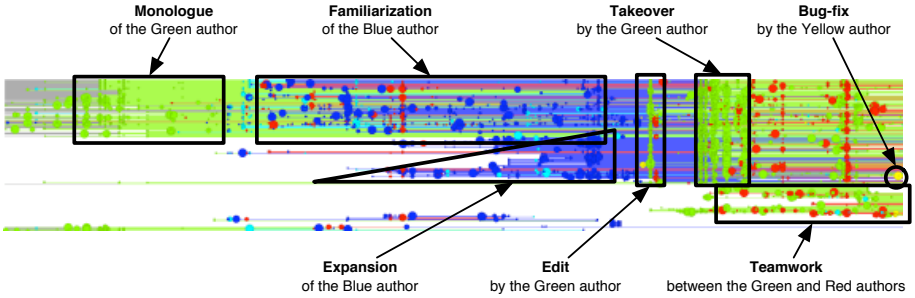


Figure 8.3: Example of the Ownership Map view. The view reveals different patterns: Monologue, Familiarization, Edit, Takeover, Teamwork, Bug-fix.

8.2.1 Ordering the Axes

Ordering the Time Axis. Subsequent file revisions committed by the same author are grouped together to form a transaction of changes that is a commit. We use a single linkage clustering with a threshold of 180 seconds to obtain these groups. This solution is similar to the sliding time window approach of Zimmerman *et al.* when they analyzed co-changes in the system [198]. The difference is that in our approach the revisions in a commit do not have to have the same log comment, thus any quick subsequent revisions by the same author are grouped into one commit.

Ordering the Files Axis. A system may contain thousands of files; furthermore, an author might change multiple files that are not near each other if we would represent the files in an alphabetical order. Likewise, it is important to keep an overview of the big parts of the system. Thus, we need an order that groups files with co-occurring changes near each other, while still preserving the overall structure of the system. To meet this requirement we split the system into high-level modules (for example the top level folders), and order inside each module the files by the similarity of their history. To order the files in a meaningful way, we define a distance metric between the commit signature of files and order the files based on a hierarchical clustering.

Let H_f be the commit signature of a file, a set with all timestamps t_{f_n} of each of its revisions f_n. Based on this the distance between two commit signatures

H_a and H_b can be defined as the modified Hausdorff distance [2] $\delta(H_a, H_b)$:

$$D(H_n, H_m) := \sum_{n \in H_n} min^2\{|m - n| : m \in H_m\}$$

$$\delta(H_a, H_b) := max\{D(H_a, H_b), D(H_b, H_a)\}$$

With this metric we can order the files according to the result of a hierarchical clustering algorithm [97]. From this algorithm a dendrogram can be built: this is a hierarchical tree with clusters as its nodes and the files as its leaves. Traversing this tree and collecting its leaves yields an ordering that places files with similar histories near each other and files with dissimilar histories far apart of each other.

The file axis of the *Ownership Map* views shown in this chapter are ordered with *average linkage* clustering and *larger-clusters-first* tree traversal. Nevertheless, our tool Chronia allows customization of the ordering.

8.2.2 Behavioral Patterns

The Overview Map reveals semantical information about the work of the developer. Figure 8.3 shows a part of the *Ownership Map* of the Outsight case study (for more details see subsection 8.3.1). In this view we can identify several different behavioral patterns of the developers:

- *Monologue.* Monologue denotes a period where all changes and most files belong to the same author. It shows on an *Ownership Map* as a unicolored rectangle with change circles in the same color.

- *Dialogue.* As opposed to Monologue, Dialogue denotes a period with changes made by multiple authors and mixed code ownership. On an *Ownership Map* it is denoted by rectangles filled with circles and lines in different colors.

- *Teamwork.* Teamwork is a special case of Dialogue, where two or more developers commit a quick succession of changes to multiple files. On an *Ownership Map* it shows as circles of alternating colors looking like a bunch of bubbles. In our example, we see in the bottom right part of the figure a collaboration between Red and Green.

- *Silence.* Silence denotes an uneventful period with nearly no changes at all. It is visible on an *Ownership Map* as a rectangle with constant line colors and no or just few change circles.

- *Takeover.* Takeover denotes a behavior where a developer takes over a large amount of code in a short amount of time, that is the developer

[2] The Hausdorff metric is named after the german mathematician Felix Hausdorff (1868-1942) and is used to measure the distance between two sets with elements from a metric space.

seizes ownership of a subsystem in a few commits. It is visible on an *Ownership Map* as a vertical stripe of single color circles together with an ensuing change of the lines to that color. A Takeover is commonly followed by subsequent changes made by the same author. If a Takeover marks a transition from activity to Silence we classify it as an *Epilogue*.

- *Familiarization.* As opposed to Takeover, Familiarization characterizes an accommodation over a longer period of time. The developer applies selective and small changes to foreign code, resulting in a slow but steady acquisition of the subsystem. In our example, Blue started to work on code originally owned by Green, until he finally took over ownership.

- *Expansion.* Not only changes to existing files are important, but also the expansion of the system by adding new files. In our example, after Blue familiarized himself with the code, he began to extend the system with new files.

- *Cleaning.* Cleaning is the opposite of expansion as it denotes an author who removes a part of the system. We do not see this behavior in the example.

- *Bugfix.* By bug fix we denote a small, localized change that does not affect the ownership of the file. On an *Ownership Map* it shows as an isolated circle in a color differing from its surrounding.

- *Edit.* Not every change necessarily fulfills a functional role. For example, cleaning the comments, changing the names of identifiers to conform to a naming convention, or reshaping the code are sanity actions that are necessary but do not add functionality. We call such an action *Edit*, as it is similar to the work of a book editor. An Edit is visible on an *Ownership Map* as a vertical stripe of unicolored circles, but in contrast to a Takeover neither the ownership is affected nor is it ensued by further changes by the same author. If an Edit marks a transition from activity to Silence we classify it as an *Epilogue*.

8.3 Validation

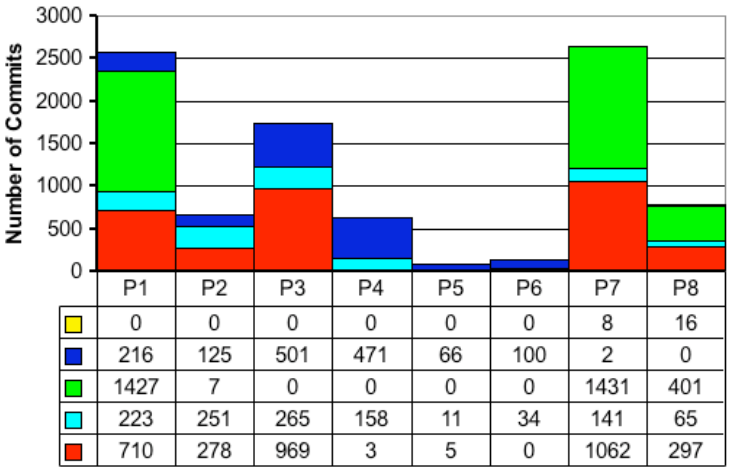
We applied our approach for several large case studies: Outsight, Ant, Tomcat, JEdit and JBoss. We report the details from the Outsight case study, and we give an overall impression on the other four well-known open-source projects.

Outsight. Outsight is a commercial web application written in Java and JSP. The CVS repository goes back three years and spans two development iterations separated by half a year of maintenance. The system is written by four developers and has about 500 Java classes and about 500 JSP pages.

Open-source Case Studies. We choose Ant, Tomcat, JEdit, and JBoss to illustrate different fingerprints systems can have on an *Ownership Map*. Ant

has about 4500 files, Tomcat about 1250 files, JEdit about 500 files, and JBoss about 2000 files. The CVS repository of each project goes back several years.

8.3.1 Outsight



	P1	P2	P3	P4	P5	P6	P7	P8
	0	0	0	0	0	0	8	16
	216	125	501	471	66	100	2	0
	1427	7	0	0	0	0	1431	401
	223	251	265	158	11	34	141	65
	710	278	969	3	5	0	1062	297

Figure 8.4: Number of commits per team member in periods of three months.

The first step to acquire an overview of a system is to build a histogram of the team to get an impression about the fluctuations of the team members over time. Figure 8.4 shows that a team of four developers is working on the system. There is also a fifth author contributing changes in the last two periods only.

Figure 8.5 shows the *Ownership Map* of our case study. The upper half are Java files, the bottom half are JSP pages. The files of both modules are ordered according to the similarity of their commit signature. For the sake of readability we use S1 as a shorthand for the Java files part of the system, and S2 as a shorthand for the JSP files part. Time is cut into eight periods P1 to P8, each covering three months. The paragraphs below discuss each period in detail, and show how to read the *Ownership Map* in order to answer our initial questions.

The shorthands in parentheses denote the labels R1 to R15 as given on Figure 8.5.

Period 1. In this period four developers are working on the system. Their collaboration maps the separation of S1 and S2: while Green is working by himself on S2 (R5), the others are collaborating on S1. This is a good example of Monologue versus Dialogue. A closer look on S1 reveals two hotspots of Teamwork between Red and Cyan (R1,R3), as well as large mutations of the file structure. In the top part multiple Cleanings happen (R2), often accompanied by Expansions in the lower part.

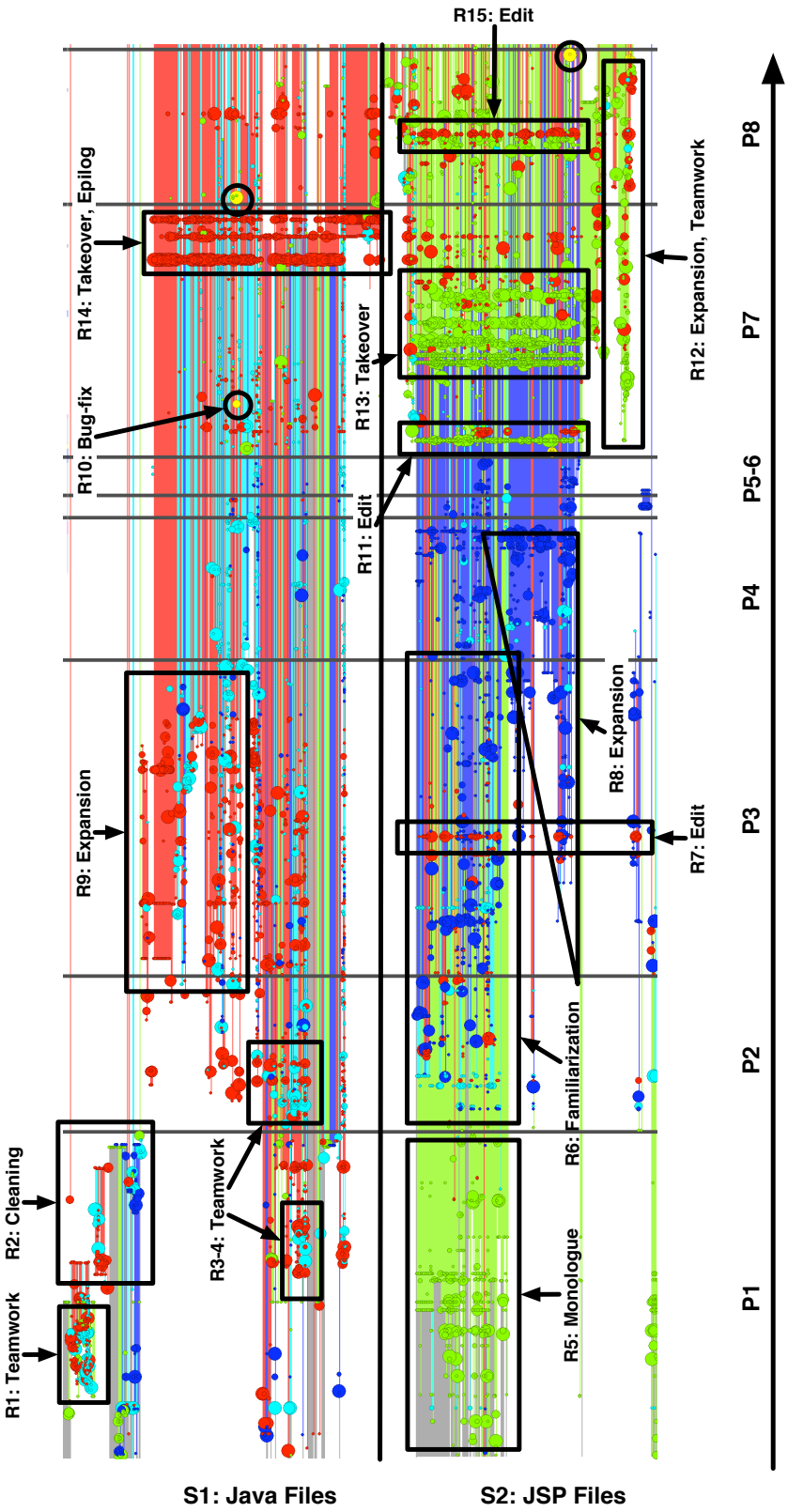


Figure 8.5: The Ownership Map of the Outsight case study.

Period 2. Green leaves the team and Blue takes over responsibility of S2. He starts doing this during a slow Familiarization period (R6), which lasts until P3. In the meantime Red and Cyan continue their Teamwork on S1 (R4) and Red starts adding some files, which foreshadow the future Expansion in P3.

Period 3. This period is dominated by a big growth of the system, the number of files doubles as large Expansions happen in both S1 and S2. The histogram in Figure 8.4 identifies Red as the main contributor. The Expansion of S1 evolves in sudden steps (R9), and as their file base grows the Teamwork between Red and Cyan becomes less tight. In contrast the Expansion of S2 evolves in small steps (R8), as Blue continues familiarizing himself with S2 and eventually takes over ownership of most files in this subsystem (R6). Also an Edit of Red in S2 can be identified (R7).

Period 4. Activity moves down from S1 to S2, leaving S1 in a Silence only broken by selective changes. Figure 8.4 shows that Red left the team, which consists now of Cyan and Green only. Cyan acts as an allrounder providing changes to both S1 and S2, and Blue is further working on S2. The work of Blue culminates in an Epilogue marking the end of this period (R8). He has now completely taken over ownership of S2, while the ownership of subsystem S1 is shared between Red and Cyan.

Period 5 and 6. Starting with this period the system goes into maintenance. Only small changes occur, mainly by author Blue.

Period 7. After two periods of maintenance the team resumes work on the system. In Figure 8.4 we see how the composition of the team changed: Blue leaves and Green comes back. Green restarts with an Edit in S2 (R11), later followed by a quick sequence of Takeovers (R13) and thus claiming back the ownership over his former code. Simultaneous he starts expanding S2 in Teamwork with Red (R12).

First we find in S1 selective changes by Red and Cyan scattered over the subsystem, followed by a period of Silence, and culminating in a Takeover by Red in the end that is an Epilogue (R14). The Takeover in S1 stretches down into S2, but it consists of a mere Edit. Furthermore we can identify two selective Bug-fixes (R10) by author Yellow, being also a new team member.

Period 8. In this period, the main contributors are Red and Green: Red works in both S1 and S2, while green remains true to S2. As Red finished in the previous period his work in S1 with an Epilogue, his activity now moves down to S2. There we find an Edit (R15) as well as the continuation of the Teamwork between Red and Green (R12) in the Expansion started in P7. Yet again, as in the previous period, we find small Bug-fixes applied by Yellow.

To summarize these finding we give a description of each author's behavior, and in what part of the system he is knowledgeable.

Red author. Red is working mostly on S1, and acquires in the end some knowledge of S2. He commits some edits and may thus be a team member being responsible for ensuring code quality standards. As he owns a good part of S1 during the whole history and even closed that subsystem at end of P7 with an Epilogue, he is the developer most knowledgeable in S1.

Cyan author. Cyan is the only developer who was in the team during all

periods, thus he is the developer most familiar with the history of the system. He worked mostly on S1 and he owned large parts of this subsystem till at end of P7. His knowledge of S2 depends on the kind of changes Red introduced in his Epilogue. A quick look into the CVS log messages reveals that Red's Epilogue was in fact a larger than usual Edit and not a real Takeover: Cyan is as knowledgeable in S1 as Red.

Green author. Green only worked in S2, and he has only little impact on S1. He founded S2 with a Monologue, lost his ownership to Blue during P2 to P6, but in P7 he claimed back again the overall ownership of this subsystem. He is definitely the developer most knowledgeable with S2, being the main expert of this subsystem.

Blue author. Blue left the team after P4, thus he is not familiar with any changes applied since then. Furthermore, although he became an expert of S2 through Familiarization, his knowledge might be of little value since Green claimed that subsystem back with multiple Takeovers and many following changes.

Yellow author. Yellow is a pure Bug-fix provider.

8.3.2 Ant, Tomcat, JEdit and JBoss

Figure 8.5 shows the *Ownership Map* of four open-source projects: Ant, Tomcat, JEdit, and JBoss. The views are plotted with the same parameters as the map in the previous case study, the only difference being that vertical lines slice the time axis into periods of twelve instead of three months. Ant has about 4'500 files with 60'000 revisions, Tomcat about 1'250 files and 13'000 revisions, JEdit about 500 files and 11'000 revisions, and JBoss about 2'000 files with 23'000 revisions.

Each view shows a different but common pattern. The paragraphs below discuss each pattern briefly.

Ant. The view is dominated by a huge Expansion. After some time of development, the very same files fall victim to a huge Cleaning. This pattern is found in many open-source projects: Developers start a new side-project and when grown up it moves to its own repository, or the side-project is terminated and removed from the repository. In this case, the spin-off is the Myrmidon project, a ceased development effort planned as successor to Ant.

Tomcat. The colors in this view are, apart from some large blocks of Silence, well mixed. The *Ownership Map* shows much Dialogue and hotspots with Teamwork. Thus this project has developers that collaborate well.

JEdit. This view is dominated by one sole developer, making him the driving force behind the project. This pattern is also often found in open-source projects: the work of a single author contributed about 80% of the code.

JBoss. The colors in this view indicate that the team underwent large fluctuations. We see twice a sudden change in the colors of both commits and code ownership: once mid 2001 and once mid 2003. Both changes are accompanied by Cleanings and Expansions. Thus the composition of the team changed twice significantly, and the new teams restructured the system.

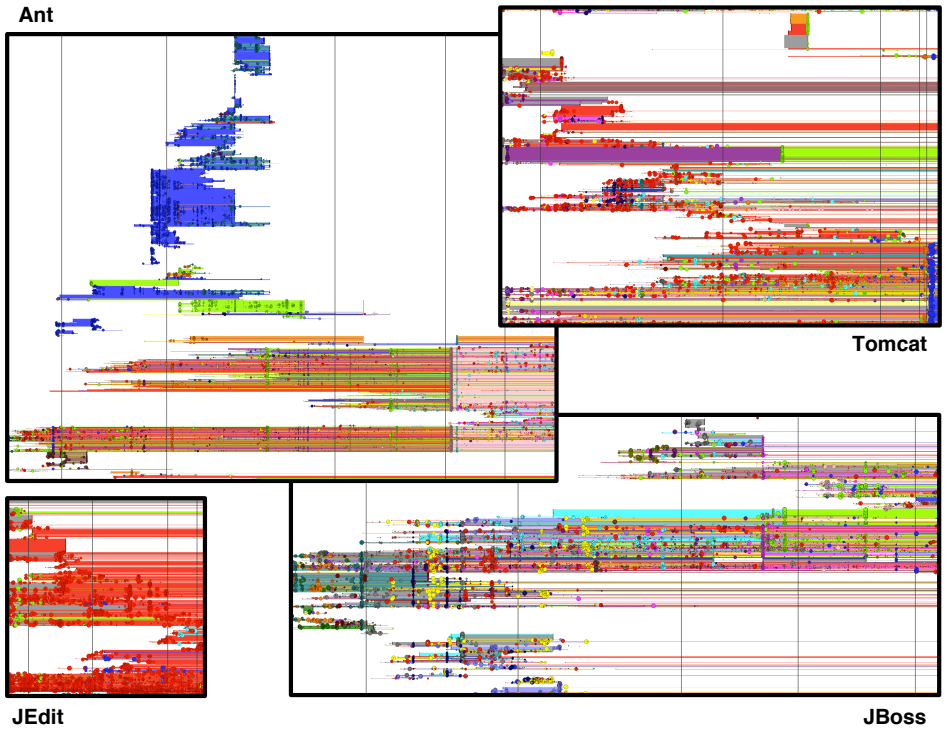


Figure 8.6: The Ownership Map of Ant, Tomcat, JEdit, and JBoss.

8.4 Discussion

On the exploratory nature of the implementation. We implemented our approach in Chronia, a tool built on top of the Moose reengineering environment [59]. Figure 8.7 emphasizes the interactive nature of our tool.

On the left of Figure 8.7 we see Chronia visualizing the overall history of the project, which provides a first overview. Since there is too much data we cannot give the reasoning only from this view, thus, Chronia allows for interactive zooming. For example, in the window on the lower right, we see Chronia zoomed into the bottom right part of the original view. Furthermore, when moving the mouse over the *Ownership Map*, we highlight the current position on both time and file axis in the panel on the right. The panel lists all file names and the timestamps of all commits. As Chronia is build on top of Moose, it makes use of the Moose contextual menus to open detailed views on particular files, modules or authors. For example, in the top right window we see a view with metrics and measurements of a file revision.

On the scalability of the visualization. Although Chronia provides zooming interaction, one may lose the focus on the interesting project periods.

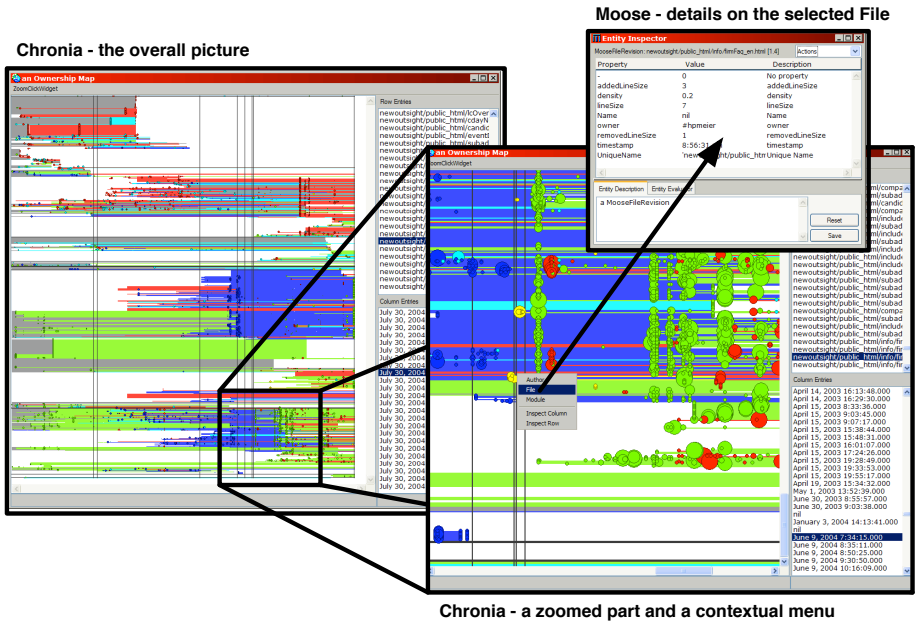


Figure 8.7: Chronia is an interactive tool.

A solution would be to further abstract the time and group commits to versions that cover longer time periods. The same applies to the file axis grouping related files into modules.

On the decision to rely on the CVS log only. Our approach relies only on the information from the CVS log without checking out the whole repository. There are two main reasons for that decision.

First, we aim to provide a solution that gives fast results; for example building the *Ownership Map* of JBoss takes 7,8 minutes on a regular 3 GHz Pentium 4 machine, including the time spent fetching the CVS log information from the *Apache.org* server.

Second, it is much easier to get access to closed source case studies from industry, when only metainformation is required and not the source code itself. We consider this an advantage of our approach.

On the shortcomings of CVS as a versioning system. As CVS lacks support for true file renaming or moving, this information is not recoverable without time consuming calculations. To move a file, one must remove it and add it later under another name. Our approach identifies the author doing the renaming as the new owner of the file, where in truth she only did rename it. For that reason, renaming directories impacts the computation of code ownership in a way not desired. More recent version control system such as Subversion and Git do not have those limitations [30].

On the perspective of interpreting the *Ownership Map*. In our visualization we sought answers to questions regarding the developers and their behaviors. We analyzed the files from an author perspective, and not from a file perspective. Thus the *Ownership Map* tells the story of the developers and not of the files for example concerning small commits: subsequent commits by a different author to one file do not show up as a hotspot, while a commit by one author across multiple files does. A pattern that we termed *Edit*.

Also from a project manager point of view the *Ownership Map* can give valuable hints. Knowing whether a developer tends more to Takeover or more to Familiarization is a good indicator to whom the responsibility of subsystem should be given. If a subsystem need rewrites and restructuring the Takeover type is a good choice, otherwise if a subsystem is a good base to be built up on the Familiarization type is a good choice.

But classifications of the authors have to be interpreted in their context. If a developer ly takes over subsystems this does not mean that he will always tend to Takeovers. In our case study (Figure 8.5) Green's Takeover of S2 in P7 must be seen in the context of the system history: Blue left the team and Green was the original developer of S2. He might have acted differently if Blue were still in the team.

Chapter 9

Discovery of Experts

Aim	Discovery of experts.
Reach	Experts who committed to local codebase.
Clues	Social (established through lexical and historical information).
Query	Fuzzy problem description given as natural language text.

Given current tool support, social clues have to be followed up through the lexical proxy of a person's name. However, often developers are facing a problem where they need help but do not know an expert by name. Maybe they are not aware that someone from their personal network is actually on expert on that matter, or because they simply do not know an expert on that matter. There is a clear need for establishing a link between fuzzy problem descriptions and experts, so that developers may follow up social clues that are otherwise out of their reach.

In this chapter we present an approach to discover experts without having to know their name. Given a problem description, such as a work item or a bug report, we provide an automated means of linking to the person with the expertise on that matter. Our work deals with assigning incoming bug reports to developers, however the techniques that we developed can be used to establish links from any fuzzy problem description to known experts. To model the expertise of developers, we require a recorded history of their changes to a system's source base. This information can be taken from a version control system. We use lexical information found in those contributions in order to model the developer's expertise.

For popular software systems, the number of daily submitted bug reports is high. Triaging these incoming reports is a time consuming task. Part of the bug triage is the assignment of a report to a developer with the appropriate expertise. In this chapter, we present an approach to automatically suggest developers who have the appropriate expertise for handling a bug report. We model developer expertise using the vocabulary found in their source code contributions and compare this vocabulary to the vocabulary of bug reports. We

evaluate our approach by comparing the suggested experts to the persons who actually worked on the bug. Using eight years of Eclipse development as a case study, we achieve 33.6% top-1 precision and 71.0% top-10 recall.

Software repositories of large projects are typically accompanied by a bug report tracking system. In the case of popular open source software systems, the bug tracking systems receive an increasing number of reports daily. The task of triaging the incoming reports therefore consumes an increasing amount of time [10]. One part of the triage is the assignment of a report to a developer with the appropriate expertise. Since in large open source software systems the developers typically are numerous and distributed, finding a developer with a specific expertise can be a difficult task.

Expertise models of developers can be used to support the assignment of developers to bug reports. It has been proposed that tasks such as bug triaging can be improved if an externalized model of each programmer's expertise of the code base is available [71]. Even though approaches for expertise models based on software repository contributions are available, existing recommendation systems for bug assignment typically use expertise models based on previous bug reports only [11, 41, 187, 144, 131]. Typically a classifier is trained with previously assigned bug reports, and is then used to classify and assign new, incoming bug reports. Our approach differs in that we train our recommendations system with all source code contributions up to the reporting date of the bug and then use the bug report's textual description as a search query against the expertise model's term-author-matrix.

In this chapter, we propose an expertise model based on source code contributions and apply in it a recommendation system that assigns developers to bug reports. We compare vocabulary found in the `diff`s of a developer's contributions with the vocabulary found in the description of a bug report. We then recommend developers whose contribution vocabulary is lexically similar to the vocabulary of the bug report.

We implemented our approach as a prototype called DEVLECT and evaluate the recommendation system using the Eclipse project as a case study. We develop and calibrate our approach on a training set of bug reports. Then we report the results of evaluating it on a set of reports of the remaining case study.

The contributions of this chapter are as follows:

- We propose a novel expertise model of developers. The approach is based on the vocabulary found in the source code contributions of developers.
- We propose a recommendation system that applies the above expertise model to assign developers to bug reports. We evaluate the system using eight years of Eclipse development as a case study.
- We report on the decay of developer expertise, observed when calibrating our approach. We apply two weighting schemes to counter this effect.

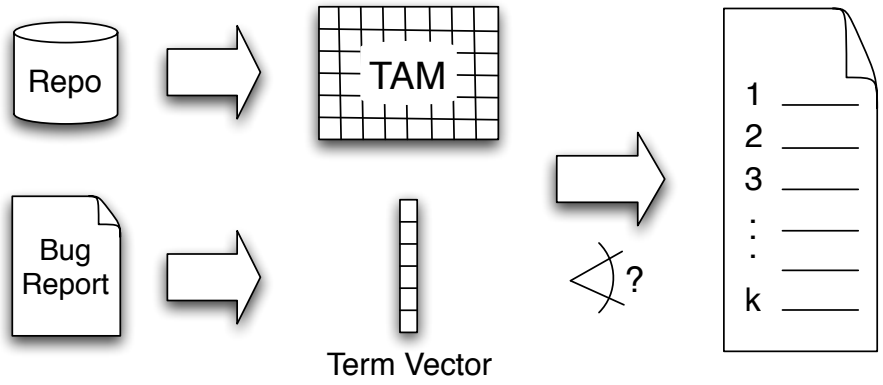


Figure 9.1: Architecture of the DEVLECT recommendation system: (left) bug reports and versioning repositories are processed to produce term vectors and term-author-matrices; (right) the cosine angle between vector and matrix columns is taken to rank developers in the suggested list of experts.

9.1 Our Approach in a Nutshell

In this chapter we present i) the construction of an expertise model of developers and ii) the application of a recommendation system that uses this expertise model to automatically assign developers to bug reports. Our approach requires a versioning repository to construct the expertise model as well as a bug tracking facility to assign developers to bug reports. We realized the approach as a prototype implementation, called DEVLECT.

In his work on automatic bug report assignment, John Anvik proposes eight types of information sources to be used by a recommendation system [10]. Our recommendation system focuses on three of these types of information: the information of the textual description of the bug (type 1), that of the developer who owns the associated code (type 6), and the information of the list of developers actively contributing to the project (type 8). We refine information type 6 to take into account the textual content of the code owned by a developer. Our recommendation system is based on an expertise model of the developer's source code contributions. For each developer, we count the textual word frequencies in their change sets. This includes deleted code and context lines, assuming that any kind of change (even deletions) requires developer knowledge and thus familiarity with the vocabulary.

Our system currently does not consider the component the bug is reported for (type 2), the operation system that the bug occurs on (type 3), the hardware that the bug occurs on (type 4), the version of the software the bug was observed for (type 5), or the current workload of the developers (type 7). Information

types 2–4 are indirectly covered, since textual references to the component, operation system, or hardware are taken into account when found in bug reports or source code. Information of type 7 is typically not publicly available for open source projects and thus excluded from our studies. Furthermore, we deliberately disregard information of type 5, since developer knowledge acquired in any version pre-dating the bug report might be of use.

Given a software system with a versioning repository, the creation of the expertise model works as illustrated in Figure 9.1:

1. For each contributor to the versioning repository, we create an empty bag of words.

2. For each contribution to the versioning repository, we create a `diff` of all changed files and count the word frequencies in the diff files. We assign the word frequencies to the contributor's bag of words.

3. We create a term-author-matrix $M_{n \times m}$, where n is the global number of words and m the number of contributors. Each entry $m_{i,j}$ equals the frequency of the word t_i in the contributions of the contributor a_j.

Given the above term-author-matrix and a bug tracking facility, the assignment of developers works as follows:

1. We count the word frequencies in the bug report and create a query vector of length n where v_i equals the frequency of word t_i in the bug report.

2. For each developer in the term-author-matrix, we take the cosine of the angle between the query vector and the developer's column of the matrix.

3. We rank all developers by their lexical similarity and suggest the top k developers.

To evaluate our approach we use Eclipse as a case study. We train our system with weekly summaries of all CVS commits from 2001 to 2008, and use the resulting expertise-model to assign developers to bug reports. We evaluate precision and recall of our approach by comparing the suggested developers to the persons who eventually worked on the bug and its report.

For example, for a report that was submitted in May 2005, we would train our system with all commits up to April. Then we would evaluate the suggested developers against those persons who worked on the bug and its report in May or later on to see if they match.

9.2 The Develect Expertise Model

Given the natural language description of a bug report, we aim to find the developer with the best expertise regarding the content of the bug report. For this purpose, we model developer expertise using their source code contributions.

Developers gain expertise by either writing new source code or working on existing source code. Therefore we use the vocabulary of source code contributions to quantify the expertise of developers. Whenever a developer writes new source code or works on existing source code, the vocabulary of mutated lines (and surrounding lines) are added to the expertise model. These lines are extracted from the version control system using the `diff` command.

Natural language documents differ from source code in their structure and grammar, thus we treat both kind of documents as unstructured bags of words. We use Information Retrieval techniques to match the word frequencies in bug reports to the word frequencies in source code. This requires that developers use meaningful names for example for variables and methods, which is the case given modern naming conventions [110].

Given a bug report, we rank the developers by their expertise regarding the bug report. The expertise of a developer is given by the lexical similarity of his vocabulary to the content of the bug report.

9.2.1 Extracting the Vocabulary of Developers from Version Control

To extract the vocabulary of a specific developer we must know which parts of the source code have been authored by which developer.

We extract the vocabulary in two steps, first building a CHRONIA model of the entire repository [74] and then collecting word frequencies from the `diff` of each revisions. The `diff` command provides a line-by-line summary of the changes between two versions of the same file. The diff of a revision summarizes the changes made to all files that changed in that revision of the repository. The identifier names and comments that appear on these lines give us evidence of the contributor's expertise in the system. Thus, we add the word frequencies in a revision's `diff` to the expertise model of the contributing developer.

Word frequencies are extracted as follows: the lines are split into sequences of letters, which are further split at adjacent lower- and uppercase letters to accommodate the common *camel case* naming convention. Next stopwords are removed (that is common words such as *the*, *and*, etc). Eventually stemming is applied to remove the grammatical suffix of words.

The comment message associated with a revision is processed in the same way, and added to the expertise model of the contributing developer as well.

9.2.2 Modeling the Expertise of Developers as a Term-Author-Matrix

We store the expertise model in a matrix that correlates word frequencies with developers. We refer to this matrix as a term-author-matrix, even though technically it is a term-document-matrix where the documents are developers. (It is commonly used in Information Retrieval to describe documents as bags of words, thus our model is essentially the same, with developers standing in for documents.)

The term-author-matrix has dimension $n \times m$, where n is the global number of words and m the number of contributors, that is developers. Each entry $m_{i,j}$ equals the frequencies of the word t_i summed up over all source code contributions provided by developer a_j.

We have found that results improve if the term-author-matrix is weighted as follows:

- *Decay of Vocabulary.* For each revision, the word frequencies are weighted by a decay factor that is proportional to the age of the revision. In the Eclipse case study, best results are obtained with a weighting of 3% per week (which accumulates to 50% per half year and 80% per annum). Please refer to Section 10.3 for a detailed discussion.

9.2.3 Assign Developers to Bug Reports regarding their Expertise

To assign developers to bug reports, we use the bug report's textual content as a search query to the term-author-matrix. Given a Bugzilla bug report, we count the word frequencies in its textual content. In particular we process both short and all long descriptions (for threats to validity see Section 10.3). We disregard attachments that are Base-64 encoded, such as attached images, as well as fields that refer to persons. From the extracted word frequencies, we create a *term vector* that uses the same word indices as the term-author-matrix. We then compute the lexical similarity between two term vectors by taking the cosine of the angle between them. The similarity values range from 1.0 for identical vectors to 0.0 for vectors without shared terms. (Negative similarity values are not possible, since word-frequencies cannot be negative either.)

We compare the term vector of the bug report with the term vectors of all developers (that is the columns of the term-author-matrix) and create a ranking of developers. For the assignment of bug reports to developers, a suggestion list of the top-k developers with the highest lexical similarities is then provided.

We have found that the results improve if the term-author-matrix is further weighted as follows:

- *Inactive Developer Penalty.* If a developer has been inactive for more than three months, the lexical similarity is decreased by a penalty proportional to the time since his latest contribution. In the Eclipse case study, best results are obtained with a penalty of 0.2 per annum. Please refer to Section 10.3 for a detailed discussion.

9.3 Case Study: Eclipse platform

To evaluate our approach we take Eclipse as a case study. Eclipse is a large open source software project with numerous active developers. Eclipse has been developed over several years now. Therefore, its version repository contains a great deal of source code developed by many different authors. Furthermore,

Eclipse uses Bugzilla as its bug tracking system, storing bug reports dating back to nearly the beginning of the project. We evaluate our results by comparing the top-k developers with the persons who eventually worked on the bug report.

Our case study covers the Eclipse project between April 22, 2001, and November 9, 2008. The source code contributions of Eclipse are stored in a CVS repository[1], the bug reports in a Bugzilla database[2]. This represents almost eight years of development, including 130,769 bug reports and 162,942 global revisions (obtained from CVS's file versions using a sliding time-window of 2 minutes [198]). During this time, 210 developers contributed to the project.

9.3.1 Setup of the Case Study

The setup of the Eclipse case study consists of two different parts. The first part is about the change database, where we use all changes before the actual bug report. The second part is about the bug database, where we make 10 partitions of which two are used in this case study. We process and evaluate both parts in weekly iterations as follows:

- We create a DEVLECT expertise model based on contributions between April 22, 2001, and the last day of the previous week.
- We generate a suggested list of the top-10 experts for all bug reports submitted in the current week.
- We evaluate precision and recall by comparing the suggestion list with the developers who, between the first day of the next week and November 9, 2008, eventually worked on the bug report.

For example, for a bug report submitted on May 21, 2005, we would train our system with all commits between April 22, 2001 and May 15, 2005, and then evaluate the list of suggested experts against the set of developers who, between May 23, 2005, and November 9, 2008, eventually handled the bug report.

We use systematic sampling to create 10 partitions of 13,077 bug reports (ordered by time) that span the entire time of the project. One partition is used as training set for the development of our approach, and another partition is used as validation set to validate our approach. We applied the approach to the validation set only after all implementation details and all calibration parameters had been finally decided on. The other partitions remain untouched for use as validation sets in future work.

In this section, we report on our results obtained on the validation partition #2. In Section 10.3 we report on results obtained from the training partition #1 while calibrating the approach.

[1] :pserver:anonymous@dev.eclipse.org/cvsroot/eclipse
[2] https://bugs.eclipse.org/bugs

9.3.2 Precision and Recall

We evaluate our approach by comparing the suggested list of experts with the developers who eventually worked on the bug report. We report on precision and recall for different sizes of suggested lists, between $k = 1$ and $k = 10$. Comparing our results to the persons who eventually worked on the bug is not optimal. For example, the person could have been assigned to the bug report by some factor other than expertise. Obtaining a better list of experts requires personal interrogation of the development team.

Precision is the percentage of suggested developers who actually worked on the bug report. Recall is the percentage of developers who worked on the bug who were actually suggested. It is typical for Information Retrieval approaches that there is a trade-off between precision and recall.

Getting the list of persons who eventually worked on a bug report is tricky. The *assigned-to* field does not always denote a person who eventually solved the bug report [187, 11, 12]. Therefore we compare our results against three configurations (C1–C3) of bug-related persons:

1. Developers who committed an actual bug fix to the software repository. For Eclipse, this information is not stored in the Bugzilla database, therefore we must rely on information from CVS commit messages. In the validation set, this information is provided for 14.3% of the bug reports only. This configuration evaluates how well we perform in suggesting experts who provide actual bug fixes.

2. Persons given by the *assigned-to* field or a *who* field of the bug report. That is, the eventual assignee (if this is a developer) and all developers who ever discussed the bug in the comment section of the report. This configuration evaluates how well we perform in suggesting experts who are capable of understanding and discussing the bug. Note that resolving a bug is not limited to providing code fixes; often the discussion is just as important to the resolution of the bug.

3. As in configuration #2, but additionally including the person identified by the *reporter* field, if the reporter is a developer, that is has a CVS login. This reflects the fact that bugs are sometimes resolved by the same people who find and track them.

Please refer to Section 10.3 for further discussion of the above configurations and their threats to validity.

9.3.3 Results

Figure 9.2 illustrates the results of the Eclipse case study. We compare lists of suggested persons of list size 1 to 10 with set of "bug related persons" as given by the three configurations (C1-3) above.

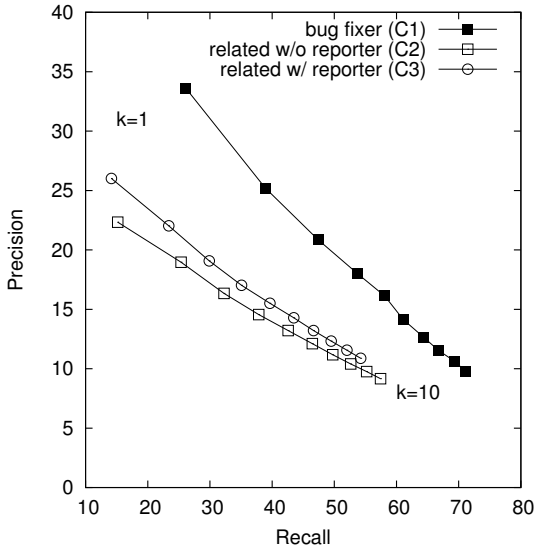


Figure 9.2: Recall and precision of the Eclipse case study: configuration C1 scores best with 33.6% top-1 precision and 71.0% top-10 recall.

The figure illustrates that recall and precision of configuration C1 are better than C2 and C3. When comparing the suggested list to the bug fixing person (C1) we achieved the best score with 33.6% top-1 precision only and 71.0% top-10 recall. Comparing of the suggested list to all related persons (C3) we score 26.0% top-1 precision and 54.2% top-10 recall. When excluding the reporter of the bug report (C2) from the suggested list scores are at 22.3% top-1 precision and 57.4% top-10 recall.

The fact that configuration C3 scores slightly better then C2 indicates that bug reporters are sometimes indeed experts regarding the reported bug and thus should be considered when triaging bug reports. We can thus conclude that an automatic assignment system should provide to the triaging person a suggested list of people that may include the reporter.

9.4 Discussion

In this section, we first discuss the calibration of our approach and then cover threats to validity.

Compared to related work, an advantage of our approach is that we do not require a record of previous bug reports. We are able to recommend developers who did not work on bugs previously. For example, we do not require that developers have worked on at least 10 resolved bug reports. On the other hand, our approach requires at least a half to one year of versioning history in order to suggest developers.

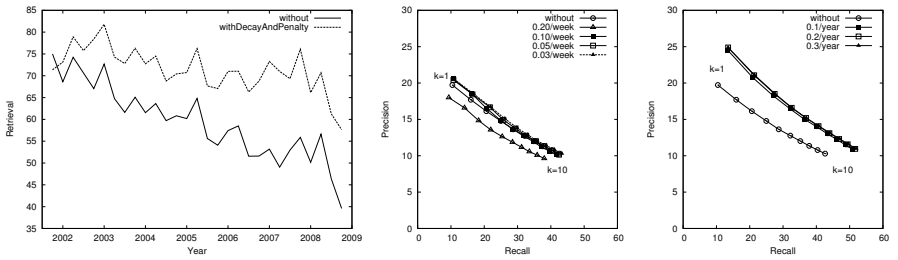


Figure 9.3: Decay of vocabulary: (left) decreasing quality of unweighted results, compared to results with *decay of vocabulary* and *inactive developer penalty* settings, (middle) precision and recall for different decay of vocabulary settings, (right) precision and recall for different inactive developer penalty settings.

One obvious threat to validity is the quality of our evaluation benchmark. We compare our suggested list against the developers who eventually worked on the bug report and assume that these are the top experts. For example, the bug report could have been assigned to a developer by some factor other than expertise. This threat is hard to counter. A better list of experts can be obtained by personal interrogation of the development team, but even this is not a golden oracle.

Another threat to validity is that we use all long descriptions, including comments, of a bug report as information sources. This may include discussions that happened after the bug has been eventually assigned or fixed, information which is not actually available when doing initial bug triage. This might impact the performance of our approach.

9.4.1 On the Calibration of Devlect

We used 1/10th of the Eclipse case study as a training set to calibrate our approach. The calibration results are summarized in Table 9.1.

The table lists top-1 precision and top-10 recall. On the first row, we list the results before calibration ($p = 19.7\%, r = 42.6\%$), and on the last row the results of the final calibration. Please note that the final results on the training set are slightly better than the results reported in subsection 9.3.3 for the validation set.

The output of the `diff` command consists of added, removed, and context lines. We experimented with different weightings for these lines (weighted `diff` in Table 1). However, we found that weighting all lines the same yields best results.

As Bugzilla bug reports consist of many fields, we experimented with different selections of fields. We found that taking short and long descriptions ("desc. fields only" in Table 9.1) yields worse results than selecting all fields except those that refer to persons, or Base64 encoded attachments.

We also experimented with Latent Semantic Indexing (LSI), an Information

Settings	Precision	Recall
reference	19.7	42.6
weighted `diff`	18.5	41.1
desc. fields only	16.7	37.7
with LSI	16.5	35.1
decay 0.03	20.5	43.2
decay 0.05	20.4	42.4
decay 0.10	20.5	41.5
decay 0.20	18.0	38.0
penalty 0.1	24.5	50.9
penalty 0.2	24.8	51.6
penalty 0.3	24.8	51.9
final calibration	26.2	54.6

Table 9.1: Summary of calibration of training set, for each settings top-1 precision and top-10 recall are given.

Retrieval technique typically used in search engines that detects polysemy and synonymy by statistical means [49]. However, we found that LSI yields poor results ("with LSI" in Table 9.1).

9.4.2 On the Decay of Vocabulary

In our experiments, we found that developer expertise decays over time. In our approach we introduced two weighting schemes to counter this effect:

- *Decay of Vocabulary.* For each revision, the word frequencies are weighted by a decay factor that is proportional to the age of the revision. Developers change their interests and by doing so change their expertise and vocabulary. To take such a shift into account, we fade the old vocabulary out bit by bit every week, so that the newest words are weighted slightly more than older ones. With time, the old words eventually fade out completely.

- *Inactive Developer Penalty.* If a developer has been inactive for more than three months, the lexical similarity is decreased by a penalty proportional to the time since his latest contribution to the software system. Inactive developers will most likely not resolve bugs anymore. In order to only recommend currently active developers (we assign bug reports during a period of eight years), developers who did not recently make a change to the software system receive a penalty.

Figure 9.3 illustrates the effect of these settings. On the left, the unbroken curve illustrates the decreasing quality of unweighted results, whereas the dotted curve shows the results obtained with weighting. Even though results improved significantly, the quality of the weighted results still slightly decreases over time. We cannot fully explain this effect; it may be due to the increasing complexity of Eclipse as a project, or perhaps the lack of mappings from CVS logins to

persons (see subsection 9.4.4) in the early years of the project impacts the results. Another cause for the trend in the most recent year, that is 2008, might be that the list of persons that worked on a bug is not yet completely known to us, which may impact the evaluation.

In the middle of Figure 9.3, precision and recall for different *decay of vocabulary* settings are given. On the right, precision and recall for different *inactive developer penalty* settings are given.

Decay of vocabulary scores best results with a weighting of 3% per week (which accumulates to 50% per half year and 80% per annum). This shows that implementation expertise acquired one year ago or earlier does not help in assigning developers to bug reports.

The inactive developer setting scores best results with a penalty of 0.2 per annum. As a result of the penalty, the matching score of a developer who has been inactive for a year is decreased. The matching scores are the lexical similarity values (between 1.0 and 0.0). Decreasing this value by 0.1 or more is typically enough to exclude a result from the top-10 list.

Interestingly, any penalty above 0.1 is better than none. The results obtained with different penalty values are almost the same. Please note, that even though the penalty removes inactive developers from the top of the suggested list, their vocabulary is not lost. The results reported for the calibration of the penalty do not make use of vocabulary decay. If a developer becomes active again, all his past expertise is reactivated as well. Thus, we use a moderate penalty of 0.2 in combination with a decay of 3% as the final calibration settings.

9.4.3 On Grouping Diffs by Week

To cope with the size of our case study, we decided to run weekly iterations rather than fine-grained iterations per bug report and revision. This reduced the time complexity from over 160,000 iterations down to 394 weekly iterations.

Grouping diffs by both author *and* week introduces the following threats to validity: If vocabulary is added and removed within the same week, it does not add to the developer's expertise. In the same way, if a file is added and removed within the same week, it is not taken into account at all. If bug reports are submitted late in the week, we might miss developers who acquired novel expertise early in the week.

If several authors worked on the same file, we cannot tell their weekly contributions apart. In this case, we weight the word frequencies by $\frac{1}{\sqrt{n}}$, where n is the number of co-developers, and assign the weighted frequencies to all co-developers. For the Eclipse case study, this applies to 3.6% of weekly file changes.

9.4.4 On other Threats to Validity

Establishing an identity relationship between CVS logins and people mentioned in bug reports is not trivial. The developer information in the CVS log is provided as a mere login name. People mentioned in a Bugzilla bug report are

listed with their email address and sometimes additionally with their first and last name. For Eclipse, the mapping between logins and active developers can be found on the Eclipse website[3]. However, the list of names of the former Eclipse developers does not include their corresponding logins[4]. We could not map 17.1% of the CVS logins and had thus to exclude 2.7% of the bug reports from our evaluation.

Information about copy patterns is not available in CVS. Bulk renaming of files appears in the change history of CVS as bulk removal of files followed by bulk addition of files. Given our current implementation of DEVLECT, this may lead to an incorrect acquisition of developer knowledge, since the entire vocabulary of the moved files is assigned to the developer who moved the files. We are thus in good shape to further improve our results by using a copy pattern detection approach [43].

[3] http://www.eclipse.org/projects/lists.php
[4] http://www.eclipse.org/projects/committers-alumni.php

Chapter 10

Credibility of Code-Search

Aim	Discovery of trustworthy projects.
Reach	Open-source projects on the internet.
Clues	Episodic (established through social and historical information).
Query	Name of an open-source project.

Searching for code examples or libraries on the internet is a common code orientation task. Developers do use code search engines to discover source code that they need in order to answer a technical question, such as implementing a given functionality. In interviews with developers, we have found that credibility is one of the major issues when copying source code from an external and thus untrusted source such as the internet. Other than internal sources, code examples taken from external sources are possibly written by an untrusted author.

We found that developers follow up social clues rather than technical issues in order to assess the trustworthiness of code search results. This is not surprising: when developers copy-paste code, they do so because they either do not have the time or do not have the expertise to technically understand the problem, thus assessing trustworthiness based on social clues as a pragmatic alternative—given the assumption that more trustworthy developers do write more trustworthy source code. For example, we have found that developers are more likely to assess a search result as trustworthy if it has been written by an author they know or by an author who belongs to a company or to an open source project that they value for its credibility. Automating this process may help reduce the time and effort that developers have to spend on following up social clues related to code search results.

The promise of search-driven development is that developers will save time and resources by reusing external code in their local projects. To efficiently integrate this code, users must be able to trust it, thus *credibility* of code search results is just as important as their relevance. In this chapter, we introduce a *credibility metric* to help users assess the trustworthiness of code search results

and therefore ease the cost-benefit analysis they undertake trying to find suitable integration candidates. The proposed credibility metric incorporates both user votes and cross-project activity of developers to calculate a *"karma"* value for each developer. Through the karma value of all its developers a project is ranked on a credibility scale. We present *JBender*, a proof-of-concept code search engine which implements our credibility metric and we discuss preliminary results from an evaluation of the prototype.

Code search engines help developers to find and reuse software. However, to support search-driven development it is not sufficient to implement a mere full text search over a base of source code, human factors have to be taken into account as well. At the SUITE 2009 workshop [108], *suitability* and *credibility* (sometimes also referred to as *trustability*) have been major issues in search-driven development, besides—of course—relevance of search results.

In this chapter we focus on the *credibility* of search results. Relevance of code search results is of course paramount, but credibility in the results is just as important. Before integrating a search result the developer has to assess its trustworthiness to take a go-or-no-go decision. A well-designed search interface allows its users to take this decision on the spot. Gallardo-Valencia *et al.* found that developers often look into human rather than technical factors to assess the credibility of search results [73]. For example developers will prefer results from well-known open source projects over results from less popular projects.

In this chapter we present a credibility metric for search results. The credibility metric is based on human factors. We use data collected from Web 2.0 platforms to assess the trustworthiness of both projects and developers. Our credibility metric is based on collaborative filtering of user votes and cross-project activity of developers. For example, if a little-known project is written by developers who also contributed to a popular open source project, the little-known project is considered to be as trustworthy as the popular project.

As a feasibility study, we implemented the credibility metric in *JBender*, a proof-of-concept code search engine. The index of our *JBender* installation currently contains credibility assessments for over 3,700 projects, based on 193,000 user votes and the cross-project activity of over 56,000 developers. In this chapter, preliminary results from an evaluation of the prototype are discussed.

Contributions of this chapter are as follows.

- We introduce a credibility metric for software projects. The credibility metric is based on human factors, and uses collaborative filtering of both user votes and cross-project activity of developers.
- We present *JBender*, a proof-of-concept implementation of our credibility metric and discuss preliminary results from an evaluation of the prototype.

10.1 Credibility Metric

In this section, we propose a credibility metric for code search results that uses collaborative filtering of both user votes and cross-project activity of developers.

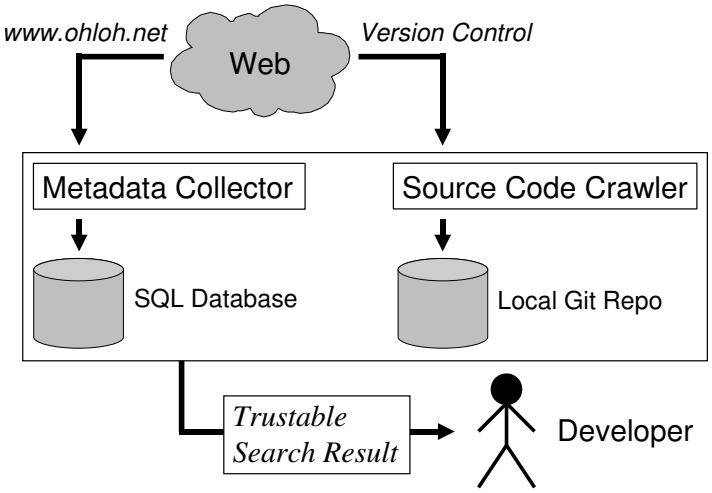


Figure 10.1: Architecture of the *JBender* prototype. *JBender* enhances search results from source code with a credibility estimate that is based on social data collected from the Ohloh Web 2.0 website.

Project Trustability	**Info**
Project: **JUnit**	**ArrayComparisonFailure** extends *AssertionError* in *org.junit.internal*

Trustability: 26.87
License: CPL
856 users
7 developers
	`private final AssertionError fCause;`
	
	`/**`
	`* Construct a new ArrayComparisonFailure with an error text and the array's`
	`* dimension that was not equal`
	`* @param cause the exception that caused the array's content to fail the assertion test`
	`* @param index the array position of the objects that are not equal.`

Figure 10.2: Screenshot of a *JBender* search result with credibility estimate. On the right there is the actual search result, with full name and code snippet. On the left there is information about the originating project and the trust value calculated by the credibility metric.

To assess the credibility of code search results we combine traditional full text search with meta-information from Web 2.0 platforms. Our credibility metric requires the following information:

- A matrix $M = (c_{d,p})$ with the number of contributions per contributor d to a project p.
- A vector $V = (v_p)$ with user votes for software projects to signal the users' trust in projects. Gallardo-Valencia *et al.* refer to user votes as "fellow users" [73].

We use collaborative filtering of both user votes and cross-project activity of developers. For example, if a little-known project is written by developers who have also contributed to a popular open source project, the little-known project is considered to be as trustable as the popular project. Since both the number of contributions per contributor and the number of votes per project follow a power-law distribution, we use *log* weighting and *tf-idf* [13] weighting where applicable.

First we define the *karma* of a contributor as

$$K_d = \sum_P w_{d,p} \log v_p \quad \text{where} \quad w_{d,p} = \frac{\log c_{d,p}}{\log \mathrm{df}(d)}$$

which is the sum of the votes of all projects, weighted by the number of contributions to these projects and divided by the inverse project frequency of the contributor (that is the number of projects to which the contributor contributed at least one contribution).

Based on this, credibility of a project is defined as

$$T_p = \sum_D w_{d,p} K_d \quad \text{where} \quad w_{d,p} = \frac{\log c_{d,p}}{\sum_{d' \in D} \log c_{d',p}}$$

which represents the sum of the karma of all the projects contributors, weighted by the number of their contributions. Note that we divide project credibility by the total number of contributions, but not contributor karma. This is on purpose, contributors are assumed to be more trustable the more they commit (based on the assumption that all accepted commits require approval of a trusted core developer, as is common in many open source projects) but projects are not per se more trustable the larger they are.

To summarize, we consider a project to be trustable if there are significant contributions by contributors who have also significantly contributed to projects (including the project in question) that have received a high number of user votes.

The proposed definition of credibility is dominated by cross-project contributors, that is contributors who contributed many times to many projects with many votes. This is in accordance with empirical findings on open source that have shown how cross-project developers are a good indicator of project success [100]. This behaviour is also known as "the rich get richer" in the theory of

Top projects (by votes)	Top Developer (by karma)
"firefox", $v_p = 7207$	"darins", $K_d = 71.97$
"subversion", $v_p = 5687$	"amodra", $K_d = 70.11$
"apache", $v_p = 5107$	"darin", $K_d = 69.09$
"mysql", $v_p = 4834$	"nickc", $K_d = 67.14$
"php", $v_p = 4081$	"Dani Megert", $K_d = 66.51$
"openoffice", $v_p = 3118$	"mlaurent", $K_d = 66.14$
"firebug", $v_p = 3109$	"Paul Eggert", $K_d = 65.89$
"gcc", $v_p = 2586$	"kazu", $K_d = 65.78$
"putty", $v_p = 2519$	"rth", $K_d = 65.25$
"phpmyadmin", $v_p = 2412$	"hjl", $K_d = 65.04$

Top projects (by credibility)
"grepWin", $T_p = 51.60$, $v_p = 32$
"GNU Diff Utilities", $T_p = 51.18$, $v_p = 645$
"Eclipse Ant Plugin", $T_p = 49.76$, $v_p = 136$
"Eclipse Java Development Tools", $T_p = 48.36$, $v_p = 647$
"Crimson", $T_p = 42.41$, $v_p = 2$
"GNU binutils", $T_p = 42.18$, $v_p = 525$
"syrep", $T_p = 42.12$, $v_p = 2$
"GNU M4", $T_p = 41.85$, $v_p = 54$
"gzip", $T_p = 41.61$, $v_p = 261$
"Forgotten Edge OpenZIS", $T_p = 40.86$, $v_p = 1$

Figure 10.3: Top ten results for A) project ranking by Ohloh, B) karma of developers, C) project ranking by trustabilty.

scale-free networks and is considered an inherent and thus common property of most social networks [21].

10.2 The JBender Prototype

We have developed a prototype, called *JBender*, which enriches code search results with credibility information. To add to the information content of search results we combine two main sources to form the *JBender* code search engine. On the one hand there is the actual code base of the search engine over which an index is created. On the other hand we have created a database of metadata for the projects in the code base.

Figure 10.1 illustrates the architecture of *JBender*. *JBender* creates a searchable index over the code base and provides a code search over it. Its novelty however lies in the underlying metadata which is linked to the projects in the searchable code base. Upon finding results from the latter *JBender* can supply the meta information stored for the result's originating project.

10.2.1 JBender's Metadatabase

Our source of meta data is the OHLOH[1] project. Ohloh is a social networking platform for open source software projects where projects (or rather their developers) can specify additional information. However Ohloh does not allow users to actually search through or interact with the source code: Ohloh is not a code search engine. Ohloh provides user contributed information on both open source projects and their developers, composing valuable information for search users. Users can vote for both projects and developers whether and how much they like them by rating projects and giving kudos to certain developers. Furthermore kudos are (automatically) given to developers who have worked for successful projects, i.e. projects with large user bases.

For the *JBender* prototype we collected the credibility meta-information from Ohloh. Metadata stored in the database includes (among others): Description of original project, project homepage, rating of the project, list of current repositories (type, url, last time of update, ...), licenses of files in the project (exact type of license, number of files), employed programming languages (percentage of total, lines of code, comment ratio, ...), the project's users and developers who worked on the project (kudos, experience, commits per project, ...).

10.2.2 JBender's Codebase

In addition to the collected metadata, *JBender* also follows the links to the version control repositories that are listed on Ohloh, creates local copies of these repositories and parses the code in Java projects to build a search index over them. *JBender* then provides a basic structured code search over various parts of the indexed source code. Examples are method/class names and their bodies, comments, visibility, dependencies and implemented interfaces.

10.2.3 Credibility enhanced results

The following data from Ohloh was directly used for the credibility metric: As contributors we used the developers of the projects and as the number of contributions we used the number of commits. As user votes we used the number of developers who "stacked" a project, which is Ohloh's terminology for claiming to be an active user of a project.[2] Thus in our case, both users and contributors are open source developers. To be a user the developers must be registered on Ohloh. This is not necessary for being a contributor, since that information is taken from version control systems.

As explained in Section 10.1 this credibility metric takes into account several of the collected meta parameters and calculates a trust metric for each result according to which the results can be sorted.

[1] http://www.ohloh.net

[2] That is, we interpret "votes" as a user expressing his trust in a project by using it.

Figure 10.2 shows a screenshot of a single search result from *JBender*. On the right there is the actual search result, with full name and code snippet. On the left there is information about the originating project and the trust value calculated by the credibility metric. Currently the raw trust measurement is displayed as a floating point number to the user. We might change that to a ranked assessment that maps the credibility to a scale from 1 to 10 to improve usability.

The layout of our search result is deliberately kept very simple and lucid in order to be efficiently usable. It has been shown that efficient search requires compact and well-arranged interfaces, which do not burden the user with too much information or a complex information seeking process [84].

10.3 Discussion

Some preliminary results Figure 10.3 illustrates the top-10 results for a) project ranking through votes by Ohloh, b) karma of developers, c) project ranking by our credibility metric. Notice how the project ranking changed through consideration of cross-project developer activity: grepWin for example has only 32 users on Ohloh but is ranked by us with top credibility because its developers are very active and have a high karma value.

Evidence of power law distribution We found that our input data (that is the user-generated data that we crawled from Ohloh) follows a power law distribution: the number of votes per project ($r = 0.95157$), the number of commits per developer per project ($r = 0.89207$), as well as number of projects per developer ($r = 0.85029$). Therefore we applied *log* and *tf-idf* weighting so that the credibility metric is not dominated by high values. At the moment project credibility ranges from zero to about 52, developer karma ranges from zero to about 72.

A note on Ohloh's kudo-rank The Ohloh website provides its own measurement of developer "karma", called *kudo-rank*. Kudo-ranks are based on a mix of user votes for projects and of user votes for developers, called *kudos*. User participation for kudos is very low and as a consequence a small clique of developers can vote themselves up to top ranks. Therefore, we decided against including kudo-ranks in our credibility function.

Possible weakness of karma ranking One must consider that developers may not use the same user names for all their commits through various repository systems. In such a case Ohloh cannot automatically collect all the developer's commits into one account; the developer would have to register and do this manually. Furthermore we blacklist commit bots. Finally the karma value could be tampered with deliberately if a user was to do a huge number of (small) commits to few highly ranked projects.

Chapter 11

Conclusion

In this last chapter we summarize the contributions made by this dissertation and point to directions for future work.

11.1 Contributions of the Dissertation

We set out to address the user needs of software engineers with regard to code navigation and code understanding. We argued that development tools need to tap unconventional information found in the source code in order to support software developers with code orientation clues that would be out of their reach without tool support.

Our key contributions are the following:

- We identified (Chapter 1) four fundamental categories of orientation clues used by developers for code navigation and code understanding: *lexical clues* referring to identifier names and concepts, *social clues* referring to a developer's personal network and to internet communities, *episodic clues* referring to personal first-hand memories of a developer, and *spatial clues* referring to the system's architecture or to source code's on-screen position as displayed by development tools.

- We introduced *Software Cartography* (Chapter 6) an approach to create spatial on-screen visualization of software systems based on non-spatial properties. Software maps are stable over time, embedded in the development environment, and can be shared among teams. We evaluated (Chapter 7) the approach in a user study, using a prototype implementation, and showed that it supports code orientation by spatial clues.

- We investigated (Chapter 3) how software engineers find answers to technical questions in a series of user studies. We found that developers typically proceed in two steps, first they narrow down their initial clue to a textual

clue, and then they query resources on the internet or local documentation for an answer.

- We presented various prototype tools that tap on unconventional information found in source code in order to provide software developers with code orientation clues that would otherwise be out of their reach. The prototypes support code orientation by providing developers with: lexical clustering of software systems (Chapter 4), summarization of software systems and their history (Chapter 5), a story-telling visualization of past contributions to a software system (Chapter 8), recommendation of experts for work items (Chapter 9), and assessing the credibility of code search results (Chapter 10).

11.2 Future Research Directions

Qualitative Studies on Developer Needs. Ethnographic research in software engineering is a rather new field. There are still many open questions regarding the user needs of developers. We investigated (Chapter 3) in a qualitative user study into how developers find answers to questions and discussed (Chapter 2) related work on developer needs, on questions that developers ask and on frequent problems of developers. Further qualitative studies on similar subjects are a promising research direction.

Hybridization of Global and Contextual Software Maps. Spatial representation of software in the development environment has received quite some attention in the past year (2010). In this dissertation, we introduced *Software Cartography* (Chapter 6), an approach that provides a global map of the entire system besides the code editor. Deline and Rowan introduced *CodeCanvas* [53], an approach that embeds code editors on a global map of the entire system. Bragdon *et al.* introduced *CodeBubbles* [35, 34], an approach that embeds code editors in contextual maps that are created on the fly. User studies of these approaches suggest that developers need both global and contextual maps. How to best address the user needs of software engineers using a hybrid approach is an open research question.

Summarization of Software Engineering Artifacts. How to best summarize code examples and work items is an open research problem. In this dissertation, we presented (Chapter 5) an approach for retrieving labels for software systems by selecting them from the vocabulary found in the source code. This approach is limited since the most descriptive umbrella terms are typically not present in the source code. How to infer the terms that best describe a source code entities is an open problem, just as is providing natural language description of source code and work items.

Example-Centric Code Search. Search-driven software engineering is a new and promising research field. Recent user studies, including the user study

presented in this dissertation (Chapter 3), have shown that developers make extensive use of internet search engines in order to find and reuse code examples. Current internet search engines, including dedicated code search engines such as *Krugle* and *Koders*, do not address this use-case. There seem to be at least three user needs to be addressed by example-centric code search: a) developers prefer code examples found on plain websites over those taken from code repositories , b) developers assess the credibility of untrusted sources based on social clues and c) developers repetitively transform code example in order to suite their local coding conventions

Story-Telling in Software Visualization. Story-telling in information visualization has received popular attention in the past year (2010). There was a one day workshop on telling stories with data (TSWD[1]) at the VISWEEK 2010 conference. In this dissertation, we presented (Chapter 8) a software visualization that tells the story of past contributions to a software system and how team members collaborated with one another. How to best support the tribal knowledge and episodic memory of development teams with story-telling visualizations is an open research question.

Social Media in Software Engineering. Most current development environments do not support code orientation by social clues. The recent rise of social media on the internet provides software engineering research with new inspiration on how to address the social needs of developers. For example, in this dissertation, we presented (Chapter 10) a preliminary approach on how to assess the credibility of code search results using collaborative filtering of user votes. Promising research directions are how to better support awareness in teams, how to encourage developers to share their knowledge with other developers and further research on how to recommend experts in social networks.

[1] http://thevcl.com/storytelling

Bibliography

[1] A. Abran, P. Bourque, R. Dupuis, and L.L. Tripp. Guide to the software engineering body of knowledge (ironman version). Technical report, IEEE Computer Society, 2004.

[2] Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from CVS. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 125–128, New York, NY, USA, 2008. ACM.

[3] 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.

[4] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *International Conference on Software Engineering (ICSE 1998)*, pages 84–93, 1998.

[5] Craig Anslow, James Noble, Stuart Marshall, and Ewan Tempero. Visualizing the word structure of Java class names. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 777–778, New York, NY, USA, 2008. ACM.

[6] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, and Andrea De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 40–49, 2000.

[7] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[8] 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.

[9] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.

[10] John Anvik. Automating bug report assignment. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 937–940, New York, NY, USA, 2006. ACM.

[11] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 2006 ACM Conference on Software Engineering*, 2006.

[12] John Anvik and Gail C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the 4th Intl. Workshop on Mining Software Repositories*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[14] S. Bajracharya and C. Lopes. Mining search topics from a code search engine usage log. In *MSR 2009*, pages 111–120, 2009.

[15] 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.

[16] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, New York, NY, USA, 2006. ACM.

[17] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of 1st Intl. Workshop on Search-driven Development, Users, Interfaces, Tools, and Evaluation (SUITE'09)*, page Too appear, 2009.

[18] 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.

[19] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[20] 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.

[21] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, reissue edition, April 2003.

[22] Olga Baysal and Andrew J. Malton. Correlating social interactions to release history during software evolution. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–597, 1995.

[24] Jacques Bertin. *Sémiologie graphique*. Les Re-impressions des Editions de l'Ecole des Hautes Etudes En Sciences Sociales, 1973.

[25] Jacques Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

[26] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 27–30, New York, NY, USA, 2008. ACM.

[27] Hugh Beyer and Karen Holtzblatt. *Contextual Design: Defining Customer-Centered Systems (Interactive Technologies)*. Morgan Kaufmann, 1st edition, September 1997.

[28] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22:36–49, October 1989.

[29] Ted J. Biggerstaff, Bharat G. Mittbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering (ICSE 1993)*. IEEE Computer, 1993.

[30] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10, 2009.

[31] 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.

[32] Grady Booch. *Object Oriented Analysis and Design with Applications*. The Benjamin Cummings Publishing Co. Inc., 2nd edition, 1994.

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

[34] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd International Conference on Software Engineering (2010)*. ACM, 2010.

[35] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512, New York, NY, USA, 2010. ACM.

[36] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *CHI 2010*, pages 513–522, 2010.

[37] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[38] 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.

[39] P. A. Burrough and R.A. McDonell. *Principles of Geographical Information Systems*. Oxford University Press, New York, 1998.

[40] 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.

[41] Gerardo Canfora and Luigi Cerulo. How software repositories can help in resolving a new change request. In *Proceedings of the Workshop on Empirical Studies in Reverse Engineering*, September 2005.

[42] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of 6th Working Conference on Reverse Engineering (WCRE 1999)*, pages 112–122. IEEE Computer Society Press, 1999.

[43] Hung-Fu Chang and Audris Mockus. Evaluation of source code copy detection methods on freebsd. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 61–66, New York, NY, USA, 2008. ACM.

[44] Mei C. Chuah and Stephen G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998.

[45] Melvin E. Conway. How do committees invent? *Datamation*, 14(4):28–31, April 1968.

[46] Davor Cubranic and Gail Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.

[47] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of 20th IEEE International Conference on Software Maintainance (ICSM 2004)*, pages 306–315, 2004.

[48] Wim De Pauw, Henrique Andrade, and Lisa Amini. Streamsight: a visualization tool for large-scale streaming applications. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 125–134, New York, NY, USA, 2008. ACM.

[49] 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.

[50] Robert DeLine. Staying oriented with software terrain maps. In *Proceedings of the 2005 International Workshop on Visual Languages and Computing*, pages 309–314. IEEE Computer Society, 2005.

[51] 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.

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

[53] Robert DeLine and Kael Rowan. Code canvas: Zooming towards better development environments. In *Proceedings of the International Conference on Software Engineering (New Ideas and Emerging Results*. ACM, 2010.

[54] Robert DeLine, Gina Venolia, and Kael Rowan. Software development with code maps. *CACM*, 53(8):48–54, 2010.

[55] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[56] Giuseppe A. Di Lucca, Massimiliano Di Penta, and Sara Gradara. An approach to classify software maintenance requests. In *Processings of 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 93–102, 2002.

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

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

[59] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[60] Stéphane Ducasse and Michele Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.

[61] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[62] Susan T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments and Computers*, 23:229–236, 1991.

[63] Susan T. Dumais and Jakob Nielsen. Automating the assignment of submitted manuscripts to reviewers. In *Research and Development in Information Retrieval*, pages 233–244, 1992.

[64] Ted Dunning. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1):61–74, March 1993.

[65] Stephen Eick, Todd Graves, Alan Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002.

[66] David Erni. Codemap—improving the mental model of software developers through cartographic visualization. Master's thesis, University of Bern, January 2010.

[67] Peter Foltz, Darrell Laham, and Thomas Landauer. Automated essay scoring: Applications to educational technology. In *Proceedings World Conference on Educational Multimedia, Hypermedia and Telecommunications (EdMedia 1999)*, pages 939–944, 1999.

[68] William Frakes and Brian Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

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

[70] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 175–184, New York, NY, USA, 2010. ACM.

[71] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 341–350, New York, NY, USA, 2007. ACM.

[72] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[73] R. E. Gallardo-Valencia and S. Elliott Sim. Internet-scale code search. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 49–52, 2009.

[74] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[75] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[76] Max Goldman and Robert C. Miller. Codetrail: Connecting source code and web resources. *JVLC*, 20(4):223–235, Aug. 2009.

[77] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, New York, NY, USA, 2008. ACM.

[78] 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.

[79] Florian S. Gysin. Improved social trustability of code search results. In *Proceedings International Conference on Software Engineering, ICSE '10, Student Research Competition*, 2010.

[80] Florian S. Gysin. Trust this Code? — improving code search results through human trustability factors. Bachelor's thesis, University of Bern, March 2010.

[81] Florian S. Gysin and Adrian Kuhn. A trustability metric for code search based on developer karma. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2010. SUITE '10.*, 2010.

[82] Florian S. Gysin and Adrian Kuhn. A trustability metric for code search based on developer karma. In *SUITE 2010*, pages 41–44, 2010.

[83] Ahmed Hassan and Richard Holt. Predicting change propagation in software systems. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Los Alamitos CA, September 2004. IEEE Computer Society Press.

[84] Marti A. Hearst. *Search User Interfaces*. Cambridge University Press, 1 edition, September 2009.

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

[86] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 79–88, New York, NY, USA, 2008. ACM.

[87] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

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

[89] Einar Hoest and Bjarte Ostvold. The java programmer's phrase book. In *Proceedings of 1st Int. Conf. on Software Language Engineering*, pages 1–10, 2008.

[90] 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.

[91] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST 2007*, pages 13–22, 2007.

[92] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE 2005*, pages 117–125, 2005.

[93] Jane Huffman-Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, January 2006.

[94] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[95] Andrew Hunt. *Pragmatic thinking and learning : refactor your "wetware"*. Pragmatic, October 2008.

[96] M. Ichii, Y. Hayase, R. Yokomori, T. Yamamoto, and K. Inoue. Software component recommendation using collaborative filtering. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 17–20, 2009.

[97] Anil Kumar Jain, M. Narasimha Murty, and Patrick Joseph Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[98] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technologies-Ada-Europe 2002*, pages 13–23, Berlin, 2002. Springer Verlag.

[99] 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.

[100] Evangelos Katsamakas and Nicholas Georgantzas. Why most open source development projects do not succeed? In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, pages 123+, Washington, DC, USA, 2007. IEEE Computer Society.

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

[102] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 184–193, 2004.

[103] 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.

[104] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, May 2007. IEEE Computer Society.

[105] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the 2004 conference on Human factors in computing systems*, pages 151–158. ACM Press, 2004.

[106] Jussi Koskinen, Airi Salminen, and Jukka Paakki. Hypertext support for the information needs of software maintainers. *Journal on Software Maintenance Evolution: Research and Practice*, 16(3):187–215, 2004.

[107] Adrian Kuhn. Semantic clustering: Making use of linguistic information to reveal concepts in source code. Master's thesis, University of Bern, March 2006.

[108] 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.

[109] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of 12th Working Conference on Reverse Engineering (WCRE'05)*, pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press.

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

[111] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software cartography: Thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution (JSME)*, 22(3):191–210, April 2010.

[112] Adrian Kuhn, David Erni, and Oscar Nierstrasz. Embedding spatial software visualization in the ide: an exploratory study. In *SOFTVIS 2010*, October 2010.

[113] Adrian Kuhn, David Erni, and Oscar Nierstrasz. Towards improving the mental model of software developers through cartographic visualization, 2010. To appear, ACM SOFTVIS 2010.

[114] Adrian Kuhn, Orla Greevy, and Tudor Gîrba. Applying semantic analysis to feature execution traces. In *Proceedings IEEE Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005)*, pages 48–53, Los Alamitos CA, November 2005. IEEE Computer Society Press.

[115] 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.

[116] Adrian Kuhn, Bart Van Rompaey, Lea Hänsenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Van Leemput. JExample: Exploiting dependencies between tests to improve defect localization. In P. Abrahamsson, editor, *Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008*, Lecture Notes in Computer Science, pages 73–82. Springer, 2008.

[117] T. Landauer and S. Dumais. The latent semantic analysis theory of acquisition, induction, and representation of knowledge. In *Psychological Review*, volume 104/2, pages 211–240, 1991.

[118] 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.

[119] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of Langages et Modèles à Objets (LMO'02)*, pages 135–149, Paris, 2002. Lavoisier.

[120] 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.

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

[122] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *ICSE 2010*, pages 185–194, 2010.

[123] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[124] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, 1986.

[125] Karl J. Lieberherr, Ian M. Holland, and Arthur Riel. Object-oriented programming: An objective sense of style. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 323–334, November 1988.

[126] Robert L. Ling. A computer generated aid for cluster analysis. *Communications of ACM*, 16(6):355–361, 1973.

[127] Erik Linstead, Lindsey Huges, Christina Lopes, and Pierre Baldi. Exploring java software vocabulary: A search and mining perspective. In *Proceedings of 1st Intl. Workshop on Search-driven Development, Users, Interfaces, Tools, and Evaluation (SUITE'09)*, page Too appear, 2009.

[128] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[129] Peter Loretan. Software cartography. Master's thesis, University of Bern, 2010. To be published.

[130] M. Lormans and A. van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE Computer Society, 2006.

[131] G. di Lucca. An approach to classify software maintenance requests. In *ICSM '02: Proceedings of the Intl. Conference on Software Maintenance (ICSM'02)*, page 93, Washington, DC, USA, 2002. IEEE Computer Society.

[132] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[133] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[134] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000)*, pages 46–53, November 2000.

[135] Andrian Marcus and Jonathan Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 125–135, May 2003.

[136] 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.

[137] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[138] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, November 2004.

[139] Amaia A. Martinez, Javier Dolado Cosin, and Concepcion P. Garcia. A metro map metaphor for visualization of software projects. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 199–200, New York, NY, USA, 2008. ACM.

[140] Dominique Matter. Who knows about that bug? — automatic bug report assignment with a vocabulary-based developer expertise model. Master's thesis, University of Bern, June 2009.

[141] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140. IEEE, 2009.

[142] 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.

[143] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of the 4th Intl. Workshop on Mining Software Repositories*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[144] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM.

[145] P. Nakov, A. Popova, and P. Mateev. Weight functions impact on lsa performance. *Proceedings of the EuroConference Recent Advances in Natural Language Processing (RANLP 2001)*, pages 187–193, 2001.

[146] Preslav Nakov. Latent semantic analysis for german literature investigation. In *Proceedings of the International Conference, 7th Fuzzy Days on Computational Intelligence, Theory and Applications*, pages 834–841, London, UK, 2001. Springer-Verlag.

[147] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, September 1993.

[148] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[149] 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.

[150] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 31–34, New York, NY, USA, 2008. ACM.

[151] Alessandro Orso, James Jones, and Mary J. Harrold. Visualization of program-execution data for deployed software. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 67–ff, New York, NY, USA, 2003. ACM.

[152] David S. Pattison, Christian A. Bird, and Premkumar T. Devanbu. Talk and work: a preliminary report. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 113–116, New York, NY, USA, 2008. ACM.

[153] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[154] Doantam Phan, Ling Xiao, Ron B. Yeh, Pat Hanrahan, and Terry Winograd. Flow map layout. In *INFOVIS*, page 29, 2005.

[155] Peter Pirolli. Information Foraging. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, chapter 205, pages 1485–1490. Springer US, Boston, MA, 2009.

[156] Peter Pirolli and Stuart Card. Information foraging in information access environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95, pages 51–58, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[157] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[158] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 37–48, Washington, DC, USA, 2007. IEEE Computer Society.

[159] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, February 2009.

[160] Paul Rayson and Roger Garside. Comparing corpora using frequency profiling. In *Proceedings of the Workshop on Comparing Corpora*, pages 1–6, October 2000.

[161] Steven P. Reiss. JOVE: Java as it happens. In *Proceedings of SoftVis 2005(ACM Symposium on Software Visualization)*, pages 115–124, 2005.

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

[163] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*, pages 317–326, 2008.

[164] M.P. Robillard. What makes APIs hard to learn? answers from developers. *Software, IEEE*, 26(6):27–34, Nov. 2009.

[165] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM.

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

[167] Edward Segel and Jeffrey Heer. Narrative Visualization: Telling Stories with Data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1139–1148, 2010.

[168] Mariam Sensalire, Patrick Ogao, and Alexandru Telea. Classifying desirable features of software visualization tools for corrective maintenance. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 87–90, New York, NY, USA, 2008. ACM.

[169] Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, 1980.

[170] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[171] Harvey Siy, Parvathi Chundi, and Mahadevan Subramaniam. Summarizing developer work history using time series segmentation: challenge report. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 137–140, New York, NY, USA, 2008. ACM.

[172] 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.

[173] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *Readings in artificial intelligence and software engineering*, pages 507–521, 1986.

[174] Frank Steinbrückner and Claus Lewerentz. Representing development history in software cities. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 193–202, New York, NY, USA, 2010. ACM.

[175] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44:171–185, 1999.

[176] 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.

[177] 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.

[178] Alexandru Telea and Lucian Voinea. An interactive reverse engineering environment for large-scale c++ code. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 67–76, New York, NY, USA, 2008. ACM.

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

[180] 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.

[181] Edward R. Tufte. *Envisioning Information.* Graphics Press, 1990.

[182] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, 2nd edition, 2001.

[183] 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.

[184] 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.

[185] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, September 2004. IEEE Computer Society Press.

[186] 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.

[187] Davor Čubranić and Gail C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[188] Fernanda Viégas, Martin Wattenberg, and Kushal Dave. Studying cooperation and conflict between authors with history flow visualizations. In *In Proceedings of the Conference on Human Factors in Computing Systems (CHI 2004)*, pages 575–582, April 2004.

[189] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 39–48, 1994.

[190] Colin Ware. *Information Visualization.* Morgan Kaufmann, 2000.

[191] 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.

[192] Richard Wettel and Michele Lanza. Visual exploration of large-scale system evolution. In *Proceedings of Softvis 2008 (4th International ACM Symposium on Software Visualization)*, pages 155 – 164. IEEE CS Press, 2008.

[193] 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.

[194] Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, November 2004. IEEE Computer Society Press.

[195] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99, Los Alamitos CA, November 2004. IEEE Computer Society Press.

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

[197] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83, Los Alamitos CA, 2003. IEEE Computer Society Press.

[198] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.