

Software Evolution Visualization

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
maandag 1 oktober 2007 om 16.00 uur

door

Stefan-Lucian Voinea

geboren te Constanta, Roemenië

Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. J.J. van Wijk

Copromotoren:
dr.ir. A.C. Telea
en
dr. J.J. Lukkien

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Voinea, Stefan-Lucian

Software Evolution Visualization / door Stefan-Lucian Voinea. -
Eindhoven : Technische Universiteit Eindhoven, 2007.

Proefschrift. - ISBN 978-90-386-1099-3

NUR 992

Subject headings: computer visualisation / software maintenance / image communication

CR Subject Classification (1998) : I.3.8, D.2.7, H.3.3

Promotor:

prof. dr. ir. J.J. van Wijk (Technische Universiteit Eindhoven)

Copromotoren:

dr. ir. A.C. Telea (Technische Universiteit Eindhoven)

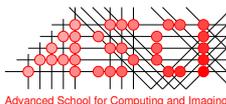
dr. J.J. Lukkien (Technische Universiteit Eindhoven)

Kerncommissie:

prof. dr. S. Diehl (Universität Trier)

prof. dr. A. van Deursen (Delft University of Technology)

prof. dr. M.G.J. van den Brand (Technische Universiteit Eindhoven)



The work in this thesis has been carried out in the research school ASCI (Advanced School for Computing and Imaging). ASCI dissertation series number: 149

©S.L. Voinea 2007. All rights are reserved. Reproduction in whole or in part is allowed only with the written consent of the copyright owner.

Printing: Eindhoven University Press

Cover design: S.L. Voinea

Front cover image: “Binary code” ©Andrey Prokhorov

Back cover image: “Cyber business” ©Emrah Türüdü

Contents

1	Introduction	1
1.1	The Software Challenge	1
1.2	Software Visualization	2
1.3	Software Evolution Visualization	5
1.4	Outline	7
2	Background	9
2.1	Introduction	9
2.2	Data Extraction	12
2.3	Reverse Engineering	13
2.4	Evolution Analysis	15
2.4.1	Requirements	16
2.4.2	Evolution Data Analysis Tools	16
2.4.3	Evolution Visualization Tools	18
2.5	Conclusions	23
3	Software Evolution Domain Analysis	27
3.1	Introduction	27
3.2	System Evolution	28
3.3	Software Evolution	33
3.4	Software Repositories	36
3.4.1	CVS	36
3.4.2	Subversion	40
3.5	Conclusions	40

4	A Visualization Model for Software Evolution	43
4.1	Introduction	43
4.2	Software Visualization Pipeline	45
4.3	Data Acquisition	46
4.4	Data Filtering and Enhancement	47
4.4.1	Selection	48
4.4.2	Metrics	49
4.4.3	Clustering	50
4.5	Data Layout	51
4.6	Data Mapping	52
4.7	Rendering	53
4.8	User Interaction	54
4.9	Conclusions	55
5	Visualizing Software Evolution at Line Level	57
5.1	Introduction	57
5.2	Data Model	58
5.3	Visualization Model	62
5.3.1	Layout and Mapping	62
5.3.2	Multiple Views	67
5.3.3	Visual Improvements	69
5.3.4	User Interaction	70
5.4	Use-Cases and Validation	73
5.5	Conclusions	76
6	Visualizing Software Evolution at File Level	79
6.1	Introduction	79
6.2	Data Model	80
6.3	Visualization Model	80
6.3.1	Layout and Mapping	81
6.3.2	Metric Views	86
6.3.3	Multivariate Visualization	87
6.3.4	Multiscale Visualization	92

6.3.5	User Interaction	97
6.4	Use-Cases and Validation	98
6.4.1	Insight with Dynamic Layouts	98
6.4.2	Complex Queries	100
6.4.3	System Decomposition	101
6.5	Conclusions	103
7	Visualizing Software Evolution at System Level	105
7.1	Introduction	105
7.2	Data Model	106
7.2.1	Data Sampling	107
7.3	Visualization Model	109
7.3.1	Layout and Mapping	109
7.3.2	Visual Scalability	111
7.3.3	User Interaction	115
7.4	Use-Cases and Validation	117
7.5	Conclusions	122
8	Visualizing Data Exchange in Peer-to-Peer Networks	125
8.1	Introduction	125
8.2	Problem Description	126
8.3	Data Model	128
8.4	Visualization Model	130
8.4.1	Server Visualization	131
8.4.2	Download Visualization	136
8.4.3	Correlation Visualization	137
8.5	Use-Cases and Validation	139
8.6	Conclusions	141
9	Lessons Learned	143
9.1	Data Acquisition and Preprocessing	143
9.2	Software Evolution Visualization	144
9.3	Evaluation	147

10 Conclusions	149
10.1 On Data Preprocessing	149
10.2 On Software Evolution Visualization	150
10.3 On Evaluation	150
10.4 Future Work	151
Bibliography	155
List of Publications	165
Summary	169
Acknowledgements	171

Chapter 1

Introduction

In this chapter we identify complexity and change as two major issues of the software industry and we introduce software evolution visualization as a promising approach for addressing them. We present the target audience of this type of visualization, the questions it tries to answer and the challenges it poses. Finding ways to design effective and efficient visualizations of software evolution is our goal and the focus of this thesis.

1.1 The Software Challenge

Software has today a large penetration in all aspects of society. According to Bjarne Stroustrup, the creator of the highly popular programming language C++,

“Our civilization runs on software” (Bjarne Stroustrup, 2003).

This penetration took place rapidly in the last two decades and continues to increase at a steady pace. However, the software industry is confronted with two increasingly serious problems.

The first problem of the software industry concerns the complexity of software. While a mid-size software application twenty years ago had a few thousands or tens of thousands of lines of code, mid-size applications nowadays have tens of millions of lines of code. Even relatively simple applications, such as the familiar Microsoft Windows Paint program, consist of tens of thousands of lines of code, spread over hundreds of files, developed by tens of people over many years. These figures are orders of magnitude larger for banking, telecom, or industrial applications. Software code can be structured in many ways, *e.g.*, as a file hierarchy; as a network of components, functions, or packages; or as a set of design patterns [49] or aspects [38, 57]. No single hierarchy suffices for understanding software, and the inter-hierarchy relations are complex. If we add dynamic and profiling data to source code, the challenge of understanding software explodes.

The second problem of the software industry is that software is continuously subject to evolution or change. The evolution of software is driven by a number of factors,

including the change of requirements, technologies, platforms, and corrective and perfective maintenance (changes for removing bugs and improving functionality). Evolution of software increases its complexity. This phenomenon is described by the so-called laws of software evolution or the increase of software entropy [70, 55]. One solution to this increasing complexity is to rewrite software systems from scratch, but the high associated costs usually prevent this. Therefore, most software projects try to keep the existing infrastructure and modify it to meet new needs. As a result, a huge amount of code needs to be maintained and updated every year (*i.e.*, the *legacy systems* problem).

An industry survey organized by Grady Booch in 2005 estimates the total number of lines of code in maintenance to be around 800 billion [14]. Out of these, 30 billion lines of code are new or have to be modified every year by about 15 million software engineers. This requires a huge amount of resources. Industry studies estimate the maintenance costs to be around 80 - 90% [40] of the total software costs, and the maintenance personnel 60 - 80% [21] of the total project staff. Studies on the cost of understanding software, such as the ones organized by Standish [100] and Corbi [24], show that this activity accounts for over half of the development effort. It is therefore utterly necessary to provide maintainers with an efficient way to take better informed decisions when planning and performing maintenance activities.

There are many possible ways to address the above challenges of the software industry, and they follow one of two main approaches (see [10]):

- *the preventive approach* tries to improve the quality of a system by improving its design and the quality of the decisions taken during the development process;
- *the assertive approach* aims to facilitate the corrective, adaptive and perfective maintenance activities, and is supported by program and process understanding and fault localization tools.

Both approaches can be facilitated by data visualization.

1.2 Software Visualization

Data visualization is the discipline that studies the principles and methods for visualizing data collections with the ultimate goal of getting insight in the data. This is reflected by one of the most accepted definitions of visualization today:

“Visualization is the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis”[53].

In his book *“Information Visualization - Perception for Design”* [120], Colin Ware summarizes the most important advantages of visualization as inferred from up-to-date research and practice:

- Visualization provides an ability to comprehend huge amounts of data;
- Visualization allows the perception of emergent properties that were not anticipated;
- Visualization facilitates understanding of both large-scale and small-scale features of the data;
- Visualization facilitates hypothesis formation.

The data visualization discipline has today two main fields of study: *scientific* and *information visualization*. While there is no clear-cut separation between the two fields, there are a number of aspects which differentiate them in practice, as follows. In scientific visualization, data is typically a sampling of continuous physical entities (*e.g.*, temperature readings acquired from a measurement or numerical simulation or tissue densities acquired from a medical scanning device). Such data has an implicit spatial encoding related to the sampling process that produced it and also typically is of numerical type. In contrast, in information visualization data is abstract in nature (*e.g.*, software artifacts, text documents, graphs, or general database tables). Such data is often not the output of some sampling process, has no natural spatial encoding, and is not of numerical type. No implicit visual encoding that maps the data to some two or three-dimensional shape exists in this case. In order to visualize the data, one must explicitly design such a visual mapping. The choice of the particular mapping used to make the abstract data visible depends on the problem and data at hand, and can greatly influence the effectiveness of visualization.

For more than a decade, scientific visualization is heavily used in many branches of mechanical engineering, chemistry, physics, mathematics, and medicine, and has become an indispensable ingredient of the scientific and engineering activity in these fields. Information visualization is a younger discipline which has started to be used in various fields of activities, including finances, medicine, engineering, and statistics. Surprisingly enough, software engineers have so far only made limited use of visualization as a tool for designing, implementing and maintaining software systems. This situation, however, is about to change.

Software visualization is a very promising solution to the complexity and evolution challenges of the software industry that supports both preventive and assertive approaches. It is a specialized branch of information visualization, which visualizes artifacts related to software and its development process.

A very good overview of software visualization and its applicability in the software engineering field is given by Stephan Diehl in his recent book "Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software" [31]. In this book, Diehl points to two surveys that investigate the perceived importance of software visualization in the software engineering community. In the first survey [68], 111 software engineering researchers were asked to give their opinion about the necessity of using visualization for performing maintenance, re-engineering and reverse engineering activities. 40% of the subjects found visualization absolutely necessary, 42% considered it is important and 7% found it relevant. Only 1% of the investigated subjects considered visualization is not important for software engineering.

In the second survey [7], the reasons for using software visualization have been investigated among 107 participants, out of which 71 came from industry and 36 from academia. The results of this survey show that the most important benefits of using visualization in software engineering are:

- Software cost reduction;
- Better comprehension;
- Increase of productivity;
- Management of complexity;
- Assistance in finding errors;
- Improvement of quality.

However, software visualization is not yet a fully accepted part of the software engineering process. According to the same study, one of the main obstructions for acceptance of software visualization by the software engineering community was the lack of integration of visualization into established tools, methodologies and processes for software development and maintenance. Another important problem of many existing software visualization methods and tools is their limited scalability with respect to the huge sizes of modern software systems.

In this thesis we address the maintenance challenge of the software industry, and we try to overcome the current limitations of software visualization. According to industry surveys [100, 24], reducing the software understanding costs is an important part of this challenge. We see two major approaches to the problem: by improving the software understanding techniques themselves to support the assertive approach, and/or by improving the decision making process which in turn will lead to a decrease in the number of performed software understanding activities, to support the preventive approach.

Both approaches can be addressed by investigating the state of the software system at a given moment in time. However, this kind of investigations provide isolated snapshots on the state of the system. While these could be sufficient to facilitate software understanding, they do not reveal the development context and trends in the evolution of the software. The presence of a development context can be useful for understanding a complex piece of software by revealing how it came into being. Software evolution trends are system specific and are useful for predictions on the state of the system. They are the basis for informed decision making during the maintenance phase.

In this thesis we try to use visualization of software evolution to get insight in the development context and in evolution trends. Our final goal is to improve both software understanding and decision making during the maintenance phase of large software projects.

1.3 Software Evolution Visualization

Software evolution visualization is a very young branch of software visualization. Software evolution visualization aims at facilitating the maintenance phase of large software projects, by revealing how a system came into being. The main question that software evolution visualization tries to answer, which is also the focus of this thesis is:

“How to enable users to get insight in the evolution of a software system?”

The intended audience of software evolution visualization consists of the management team and software engineers involved in the maintenance phase of large software projects. These professionals usually face software in the late stages of its development process, and need to get an understanding of it, often with no other support than the source code itself. In software engineering, one does not speak of different persons involved in the software maintenance process, but of different roles. The same role can be played by different persons, and the same person can play several roles at a single moment or different moments during the lifetime of a software project. The most common roles targeted by software evolution visualization and the potential benefits are summarized below:

- project managers can get an overview of source code production and use identified trends as support for decision making;
- release managers can monitor the health of a given product evolution and decide when it is ready for a new release;
- architects can identify subsystems needing redesign or suffering from architectural erosion;
- testers can identify the regression tests required at system migration;
- developers can get familiar with the software and set-up their social network based on relevant technical issues (*e.g.*, by identifying the developers that previously worked on the same piece of source code).

For all these roles, software evolution visualization tries to answer a number of questions, following the visual analytics mantra: “detect the expected and discover the unexpected” [107]. These questions range from concrete, specific queries about a certain well-defined aspect or component of a software system, to more vague concerns about the evolution of the system as a whole. Typical questions are:

- What code was added, removed, or altered? When? Why?
- How are the development tasks distributed among the programmers?
- Which parts of the code are unstable?
- How are source code changes correlated?
- What are the project files that belong and/or are modified together?

- What is the context in which a piece of code appeared?
- How difficult to maintain is the system?

A number of challenges have to be met, in order to turn software evolution visualization into an effective instrument for the software engineer. Some of these challenges are common to data visualization. Some other challenges are specific to the context of the software engineering industry in general, and to the context of software evolution in particular. All in all, these challenges relate to the ultimate goal of any visualization, that is, to support the user to solve a specific problem in an efficient manner. Among the challenges of software evolution visualization, the following are worth mentioning:

- *scalability*: Modern software systems are huge. Visualizing not just a single snapshot, but an entire evolution of such a system, is a daunting task. First, this requires the analysis of a huge amount of information, which has to be done efficiently to facilitate interactive or near-interactive analysis and discovery. Second, the results of the analysis must be displayed in an efficient manner. If the datasets at hand are too large, one might consider presentation on large displays or multi-screen configurations. However, in the typical software engineering context, it is more realistic to assume the user must work with single-screen commodity graphics displays. This brings the problem of efficient and effective display of a large information space on a limited rendering real estate.
- *intuitiveness*: Software related artifacts and entities, such as files, lines of code, functions, modules, programmers, bugs, and releases, are abstract entities interconnected by a complex network of relations. Designing appropriate visual representations that are easy to follow and effectively convey insight into this high-dimensional data space is one of the largest challenges of software evolution visualization.
- *usability*: Software understanding is a dynamic and repetitive process which requires many queries of different (interrelated) aspects of the software corpus. Typically, users formulate a hypothesis and consequently they try to validate it. In this process they might discover new facts that lead to changes of the hypothesis and require new validation rounds. Designing software evolution visualization applications with the requirements and specifics of the user activities in mind is crucial for success.
- *integration*: To be successful in the long run, but also simply to be accepted, software visualization applications must be seamlessly integrated with the established tools of the trade of the software engineering process, such as code analyzers, compilers, debuggers, and software configuration management systems. This requires a careful design and architecture of the visualization tools.

Besides these challenges of software evolution visualization, many other challenges exist as well. Specific software development contexts, *e.g.*, the use of a particular programming language or development methodology, may require the design of customized interactive visual techniques and tools. If software evolution visualization is to target

large projects, facilities must be developed to support collaborative work of several users, possibly at different locations. Finally, software evolution visualizations should target questions and requirements of a wide range of users, from the technically-minded programmers to the business and process-oriented managers. All these constraints pose a formidable challenge, and open novel research grounds to software evolution visualization.

1.4 Outline

The remainder of this thesis is organized as follows:

Chapter 2 positions the thesis in the context of related research on analysis and visualization of software evolution.

In Chapter 3, an analysis of the software evolution domain is performed to formalize the problems specific to this field. To this end, a generic system evolution model and a structure based meta-model for software descriptions are proposed. Consequently, these models are used to give a formal definition of software evolution. Challenges of using this description with empirical data available from current software evolution recorders are addressed.

In Chapter 4 a visualization model for software evolution is proposed based on the software evolution model introduced in Chapter 3. The visualization model consists of a number of steps with specific guidelines for building visual representations of software evolution.

Chapters 5, 6 and 7 present three applications that make use of the visualization model proposed in Chapter 4 to support real life software evolution analysis scenarios. These applications cover some of the most commonly used software description models in industry: file as a set of code lines, project as a set of files, and project as one software unit. In agreement with the addressed models, the presented applications visualize software evolution at line, file and respectively system level. For each application, relevant use cases are formulated, specific implementation aspects are presented, and results of use case evaluation studies are discussed.

In Chapter 8, a novel visualization of data exchange processes in Peer-to-Peer networks is proposed. The aim of presenting this visualization is twofold. First, we illustrate how to visualize time dependant software-related data other than software source code evolution. Secondly, we show that the visual techniques that we have developed for software evolution assessment can be put to a good use for other applications as well.

Chapter 9 contains an inventory of reoccurring problems and solutions in the visualizations of software evolution discussed in the previous chapters. Generic issues that transcend the border of the software evolution domain are also identified and presented together with a set of recommendation for their broader applicability.

Eventually, Chapter 10 gives an overview on the main contributions and findings of the work presented in this thesis. It also outlines remaining open issues, and possible research directions that can be followed to address them.

Chapter 2

Background

In this chapter we first describe the position of software evolution analysis in software engineering. Next, we give a number of requirements for an ideal tool to support software evolution analysis. Finally, we give an overview of related work in the area of designing such tools, with an emphasis on visualization.

2.1 Introduction

Software engineering (SE) is a relatively new discipline (*i.e.*, firstly mentioned by F.L. Bauer in 1968 [86]) that tries to manage the ever increasing complexity of designing, creating, and maintaining software systems. To this end it applies technologies and practices from many fields, from computer science, project management, engineering, interface design to application specific domains.

The traditional software engineering pipeline consists of an extensive set of activities which covers the complete lifetime of a software product, from its creation to the moment the product gets discontinued. These activities are, in chronological product lifetime order [78]:

1. product and user requirement gathering;
2. software requirements gathering;
3. construction of the software architecture and design;
4. implementation of the software product;
5. testing and releasing;
6. deployment;
7. maintenance;
8. discontinuation (end of life).

The first six phases, from requirement gathering up to and including deployment, are traditionally called the *forward engineering* process. The forward engineering process is sketched in the upper part of Figure 2.1, which gives an overview of the traditional SE pipeline consisting of forward engineering and maintenance activities. In this figure, rounded rectangles represent activities, such as requirement gathering, implementation, or maintenance actions, and sharp corner rectangles represent artifacts which are the typical input and output for activities, such as software source code, documentation, metrics, but also maintenance decisions. The figure is structured along two axes: Phases of the SE process (vertical) and types of activities involved (horizontal).

After the first version of the software product is released and deployed, software enters the *maintenance* phase (Figure 2.1). This is typically the longest and most resource consuming phase. Finally, the software product lifecycle ends with the discontinuation of the product. The software itself can be used afterwards as well, but there are no more development or maintenance resources invested.

As explained in Chapter 1, the maintenance phase can last for many years, involve a wide range of individuals, and take a major share of the resources allocated to the overall software engineering process. To find efficient ways to support this phase is, therefore, a major concern of the software engineering community. In this thesis we propose a novel approach addressing this concern. Consequently, we shall next focus only on the maintenance part of the software engineering process, and not further detail the forward engineering part.

The maintenance phase (Figure 2.1) can be split in four parallel tracks depending on the type of activities that take place (see [10]). These tracks and the corresponding activities are:

1. *corrective maintenance*: remove bugs from the software;
2. *adaptive maintenance*: adapting the software to new environments;
3. *perfective maintenance*: add features and overall improve the software;
4. *preventive maintenance*: change the software to facilitate further evolution.

In the corrective maintenance track, activity is typically triggered by the occurrence of development problems such as detection of bugs in the existing code. Adaptive maintenance is required to port the system to new software or hardware platforms. Perfective maintenance takes place when software requirements change and system functionality has to be altered. Preventive maintenance is typically triggered by the need to reduce the time between releases and to facilitate further evolution of the software product.

Ideally, maintenance activities and their outcome should be reflected in the project support documentation. However, a characteristic phenomenon that is typical to software evolution is that the structured information which is originally available on the software system, consisting of requirements, functional documentation, architectural and design documents, and commented source code, quickly gets degraded during the maintenance process. A typical example is that of paper documents getting out-of-sync with the source code. In the vast majority of projects, source code plays an essential and particular role

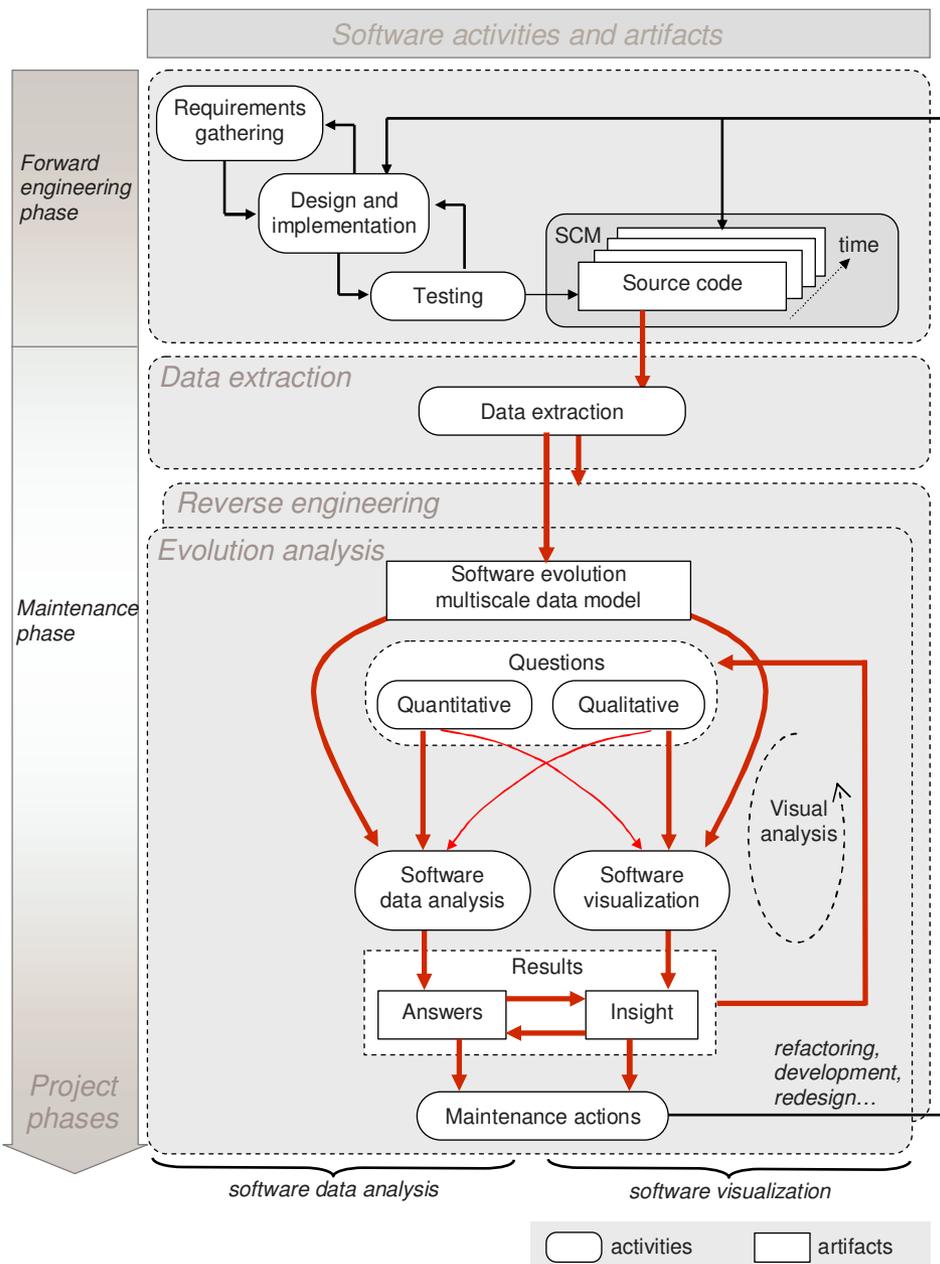


Figure 2.1: Evolution analysis in the maintenance phase of software projects

in the maintenance phase, since it is the critical item that has to be maintained, and also the only up-to-date item at any moment in time. This observation has been succinctly captured by Stroustrup in his statement that "source code is the main asset and currency of the software industry" [103]. Hence, actions in the maintenance phase usually start with an *analysis* of the available source code.

In most cases, the source code is available in its latest version, but also in all intermediate versions, via so-called *software configuration management* (SCM) systems, such as CVS [28] and Subversion [104]. These systems maintain databases, also called *repositories*, which store the evolution of a number of software artifacts in digital form (*e.g.*, source code, documents, datasets, bug and change reports). The main functionality of the SCM system is to maintain the most up-to-date version of each stored artifact. Users can update artifacts by first checking them out from the repository, performing changes, followed by checking them in. Efficient storage schemes are developed to minimize the space needed, for instance, by recording only the incremental changes to a given artifact. In most cases, SCM systems support hierarchical file-based structures (directory trees) as artifacts. In such cases the smallest unit of configuration management is a file. Typical SCM systems offer facilities to support a multi-user, multi-site paradigm where several users can modify the same set of artifacts remotely from different locations.

SCM systems provide the "raw material" that the maintenance activities work on. However useful in storing the source code and its changes, SCM systems do not give immediate answers to maintenance related questions like, for example, "why a certain change took place" or "what are the consequences or implications of a given change". Also, SCM systems often store change information on a too low level. Indeed, as the aim of most SCM systems in use nowadays is to efficiently store and retrieve changes of textual or binary data contained in various files, their change information representation is geared towards this end. For example, SCM systems can tell a user quite easily which lines of text have changed in a certain version of some source code text file, but not what the changes are at function or software subsystem level. Hence, the first phase of a typical maintenance activity is to analyze a given SCM repository in order to distill higher-level, task-specific information from the low-level recorded changes. To do this, we must first have access to the repository information itself. After this low-level information is available, higher-level information can be distilled to be used in driving the maintenance activities.

We detail several directions of previous work related to our goal of getting visual insight into evolving software. In Section 2.2, we discuss the process of extracting data from SCM systems. The relation between understanding software evolution and the reverse engineering discipline is discussed next in Section 2.3. Section 2.4 zooms in two important current approaches in the process of software evolution analysis: evolution mining and evolution visualization. Finally, Section 2.5 concludes this chapter.

2.2 Data Extraction

The first step that is necessary to analyze the evolution of a software system is to have access to the low-level facts stored in SCM repositories. Although this step is critical in

obtaining the right data for further processing, this operation is not supported at a fully appropriate level in practice. As a result, data extraction requires considerable effort and is often system specific. For example, many researches target CVS [28] repositories, given their large popularity and free availability on the market *e.g.*, [45, 48, 51, 73, 128, 131]. Yet, there exists no standard application programming interface (API) for CVS data extraction. Many CVS repositories are available over the Internet, so such an API should support remote repository querying and retrieval.

A second problem is that CVS output is meant for human, not machine reading. Many actual repositories generate ambiguous or non-standard formatted output. Several libraries provide an API to CVS, such as the Java package JavaCVS [63] and the multi-language module LibCVS [71]. However, JavaCVS is undocumented, hence of limited use, whereas LibCVS is incomplete as it does not support remote repositories. The Eclipse environment implements a CVS client [34], but does not expose its API. The Bonsai project [13] offers a toolset to populate a database with data from CVS repositories. However, these tools are more a web access package than an API and are little documented. The only software package we found that offers a mature API to CVS is NetBeans.javacvs [87]. It allegedly offers a full CVS client functionality and comes with reasonable documentation. Although we did not run comprehensive evaluation tests on this package, it appeared to be the best alternative for implementing an API controlled connection with a CVS repository. In contrast, low-level procedural access to Subversion [104] repositories is better supported by cleaner and better documented APIs, a fact which can be ascribed to the fact that Subversion is a newer, more sophisticated system than CVS.

Concluding, although low-level data access can be seen as an implementation detail, the availability of a robust, efficient, well-documented, usable mechanism to query a SCM repository is not a granted fact. The availability of such a mechanism can largely influence the design and success of supporting tools, as well as the fulfillment of the seamless integration requirement of analysis and software management tools (Chapter 1).

2.3 Reverse Engineering

To support a wide range of maintenance scenarios, the analysis activities must extract a wide range of types of information from a given repository. This information exists at higher levels than what is provided via the APIs of current SCM systems. Indeed, typical SCM systems, such as CVS [28] or Subversion [104] are content neutral. That is, they do not make any assumptions about what types of artifacts are checked in the system beyond the level of files made of lines or bytes. This makes these systems, on the one hand, very generic and applicable to a large class of problems. On the other hand, maintenance activities take place at many more levels besides the file level. To perform such activities, additional analysis is necessary to:

1. derive various types of facts from the stored files;
2. determine which of the extracted facts have changed, and how.

The first activity mentioned above is the subject of the sub-discipline of software engineering called *reverse engineering* [18, 119, 12, 6]. Given a set of weakly structured software artifacts, such as the files stored in a SCM repository, reverse engineering is concerned with the task of extracting various facts about the software stored in those files. These facts exist on a wide range of levels of abstraction, and are useful for several maintenance activities.

A first example of facts concerns the structure of the software. Here, the relevant information to be extracted is typically one-to-one with the original program structure, and consists of, for instance, lines of code, functions or methods, classes, namespaces, packages or modules, subsystems, and libraries. This type of analysis is also called static program analysis. Many tools have been developed that can be used in extracting structural facts from source code [3, 23, 26]. These tools are known under various names, such as parsers and fact extractors, and can deliver amounts of information ranging from a simple containment hierarchy of the main constructs of the code (*e.g.*, files and functions) to a fully annotated syntax tree (AST) of the source code containing the semantics of every single token in a file. Besides analyzing the source code, fact extractors can also generate different types of structural information, *e.g.*, UML class diagrams, message sequence charts, or call graphs from the source code. Structural fact extraction and code parsing is a wide area of research with decades of experience, which we shall not detail further in this context. Overviews are given in [90, 122, 109]. Moreover, most research in this area has targeted the analysis of single versions of a software system.

A second example of facts that can be extracted from the source code concern the quality of the software. Here, the relevant information to be extracted is not necessarily one-to-one with the original program structure, but consists of a number of quantitative or qualitative metrics. These can be computed at various levels of granularity, ranging from lines of code to entire subsystems, and are useful in signaling the occurrence of specific situations. For example, high values of a coupling strength metric can indicate a monolithic, less modular, system which may be inflexible during a longer maintenance period.

In the above, we have considered both structural and metric facts extracted from single versions of a software system. If we combine the structural and metric information extracted from a given system version, we obtain a so-called *multiscale* dataset, *i.e.*, a representation of the software at several levels of detail, and from several perspectives. Although useful in assessing maintenance issues related to a single system version, such information cannot answer questions that involve several versions. For example, if we had the appropriate tools, we could use this information to answer the question "is a given software version unstable?", but not "is the system evolving towards an increasingly unstable state?". Such questions are important for preventive maintenance, when one must detect an evolutionary trend and perform maintenance before the actual undesired situation occurs.

In the following section, we review a set of tools and techniques mentioned in literature that are currently available for extracting and analyzing information on the evolution of software systems. These tools and techniques are complementary to, and not replacing, the static analysis tools for reverse engineering discussed. While static analysis tools give a wealth of information about a concrete version but do not look at the greater picture

of evolving software, evolution analysis tools focus on uncovering the dynamic, time-dependent trends in a software project, but provide less detail on the structure and metrics of each particular version.

The focus of this thesis being visualization, let us mention that both structural and metric information extracted from a single version can be visualized in various ways and at various levels of detail. Call graphs can be displayed using ball-and-stick diagrams and matrix plots to uncover system structure and assess modularity [102, 1]. Source code can be displayed annotated with metrics to emphasize the exact location of various desired or undesired events [72]. Metrics can be combined with UML diagrams extracted from source code to correlate system quality and architecture [106, 105]. All these techniques are covered by the traditional software visualization discipline, for which good overviews can be found in [101, 31]. Our specific interest area being software evolution visualization, we shall further detail (in Section 2.4.3) only those visualization techniques that target change in software systems.

2.4 Evolution Analysis

As explained in the previous section, the analysis step of the maintenance phase involves both single-version analysis and multi-version, or evolution, analysis tools. In this section, we review techniques and tools that focus on analyzing software evolution.

There exist two major approaches towards analyzing the evolution of software systems: *data analysis* and *data visualization* (Figure 2.1).

Data analysis uses a number of data processing activities to find answers to specific questions regarding the evolution of software, but also to mine the data and discover new aspects that improve the understanding of a system. Examples of data analysis functions are the computation of search queries, software metrics, pattern detection, and system decomposition, all familiar to reverse engineers [5, 45, 52, 129].

The goal of the data visualization approach is also twofold. On the one hand it tries to address specific questions with answers that are not simple to encode in figures or words (*e.g.*, “how are the maintenance activities distributed over the team”). On the other hand, visualization tries to give deeper insight into vague problems, which can in turn lead either to unexpected answers or to formulation of more specific questions.

These two approaches correspond closely to the main activities performed during data extraction and analysis of software evolution (Figure 2.1) using tool support. Data analysis tools try to apply specific algorithms on extracted evolution data. Visualization tools try to use the human vision system both to give insight in data and to answer specific questions. Unfortunately, most existing tools tend to focus exclusively on one of the above categories (see Table 2.1). This leads in practice not only to a weak acceptance of software evolution tools, but also to a slow progress in developing and perfecting of the category specific activities and techniques. We next present a number of requirements that tools targeting software evolution should address in order to overcome these limitations, followed by an overview of the state of the art in data analysis and visualization for software evolution.

2.4.1 Requirements

The requirements presented below attempt to integrate in one tool all previously identified data evolution analysis activities. To this end they detail the high-level usability, scalability, intuitiveness and integration requirements that we set for software visualization tools at the end of Chapter 1, for the specific context of maintenance activities. All in all, an ideal tool that supports the analysis process in Figure 2.1 should address the following aspects:

- **(R1)** multiscale: able to query/visualize software at multiple levels of detail (lines, functions, packages);
- **(R2)** scalability: handle repositories of thousands of files, hundreds of versions, millions of lines of code;
- **(R3)** data mining and analysis: offer data mining and analysis functions such as queries and pattern detection;
- **(R4)** visualization: provide visualizations that effectively answer specific questions as well as offer deeper insight;
- **(R5)** integration: the offered services should be tightly integrated in a coherent, easy-to-use tool.

Table 2.1 summarizes some of the most popular evolution analysis and visualization tools in the three categories discussed above. In the next section, evolution data mining (Section 2.4.2) and evolution visualization tools (Section 2.4.3) are discussed in more detail.

2.4.2 Evolution Data Analysis Tools

Evolution data analysis is a relatively new direction of research. Few methods have been proposed to offer access to higher level aggregated information about the project evolution. Fischer *et al.* [45] have proposed a novel method to extend the evolution data contained in the SCMs with information about file merge points. Additionally, they have presented the benefits of integrating SCM evolution data with specific information about bug tracking. Sliwerski *et al.* [97] have proposed a similar integration to predict the introduction of defects in code.

One of the subjects more extensively addressed by the research community is the recovery of SCM transactions. Gall [48], German [52] and Mockus [82] have proposed transaction recovery methods based on a fixed time windows. Zimmermann and Weißgerber [130] built on this work, and have proposed better mechanisms that involve sliding windows and information acquired from commit e-mails.

Another issue that has been investigated is the use of history recordings to detect logical couplings. Ball [5] has proposed a new metric for class cohesion based on the SCM extracted probability of classes being modified together. Relations between classes

Tool Name	Data Extraction	Reverse Engineering	Evolution Analysis Activities	
			Data Visualization	Data Analysis
LibCVS [71]	X			
WinCVS [123]	X			
JavaCVS [63]	X			
Bonsai [13]	X			
Eclipse CVS plugin [34]	X			
NetBeans.javacvs [87]	X			
Release History Database [45]	X	X		X
Diff [32]		X		X
eRose [131]	X	X		X
QCR [48]		X		X
Social Network Analysis [73]		X		X
MOOSE [33]	X	X		
Historian [59]	X		X	
SeeSoft [37]		X	X	
Augur [47]	X		X	
Xia [126]		X	X	
WinDiff [124]			X	X
Hipikat [27]	X		X	X
Gevol [22]		X	X	
VRCS [67]	X		X	
3DSoftVis [93]		X	X	
Evolution Matrix [69]			X	
Evolution Spectograph [125]	X		X	
RelVis [91]			X	
SoftChange [51]	X		X	X
EPOSee [15]			X	

Table 2.1: Software evolution analysis tools: activities overview

based on the change similarities have been proposed also by Bieman *et al.* [11] and Gall *et al.* [48]. Relations between finer grained building blocks, like functions, have been addressed by Zimmermann *et al.* [129, 131] and by Ying *et al.* [128].

The presence of user information in the SCMs has been used to assess developer networks. Lopez-Fernandez *et al.* [73] have applied general social network analysis methods on the information stored in SCMs to characterize the development process of industry size projects and find similarities between them. Ohira *et al.* [89] have exploited the user information stored in SCMs to build cross process social networks for easy sharing of knowledge.

Concluding, compared to other fields of software engineering, such as reverse engineering, software evolution data analysis is a less explored direction of research. However, evolution data analysis tools are promising instruments for understanding software and its development process. By integrating in these tools history recordings with other sources of information such as bug tracking systems and developer e-mails, the analysis accuracy can be improved and a broader range of usage scenarios can be dealt with.

2.4.3 Evolution Visualization Tools

Evolution visualization takes a different approach to software evolution assessment than evolution data analysis. The focus is on how to make the large amount of evolution information available to the user, and let the user discover patterns and trends by himself. A rather small number of tools have been proposed in this direction.

SeeSoft [37] is one of the first visualization tools we are aware of that addresses software evolution analysis. It uses a direct “code line to pixel line” mapping and color to show code fragments corresponding to a given modification request. Using a similar approach, *Augur* [47] is a more recent tool that combines in a single image information about artifacts and activities of a software project at a given moment (see Figure 2.2). Both *SeeSoft* and *Xia* [126] use treemap layouts to show software structure, colored by evolution metrics (see Figure 2.3), *e.g.*, change status (*SeeSoft*), time and author of last commit and number of changes (*Xia*).

Such tools, however, focus on revealing the structure of software systems and uncover change dependencies only at single moments in time. They do not show code attribute and structural changes made during an entire project. Evolution overviews allow discovering that problems in a specific part of the code appear after another part was changed. They also help finding files having tightly coupled implementations. Such files can be easily spotted in a temporal context as they most likely have a similar evolution. In contrast, lengthy manual cross-file analysis activities are needed to achieve the same result without an evolution overview.

As a first step towards global evolution views, UNIX’s *gdiff* [32] and its Windows version *WinDiff* [124] show code differences (insertions, deletions, and modifications) between two versions of a file (see Figure 2.4). *Hipikat* [27] is a similar tool that enriches the information regarding version differences with context specific information recorded during the project such as bug reports or e-mails. This information appears to be very useful in understanding changes across versions. However effective for comparing pairs



Figure 2.2: Code line to pixel line visualizations: (a) color encodes the ID of a modification request (SeeSoft [37]); (b) color encodes the ID of the version when the corresponding code changed for the last time (Augur [47]).



Figure 2.3: Software visualization using treemaps to encode structure and color to encode evolution metrics: (a) color encodes the change status of code: gray = unmodified, green = added, red = deleted, yellow = changed (SeeSoft [37]); (b) color encodes the commit date of last revision: green = old, blue = recent (Xia [126]).

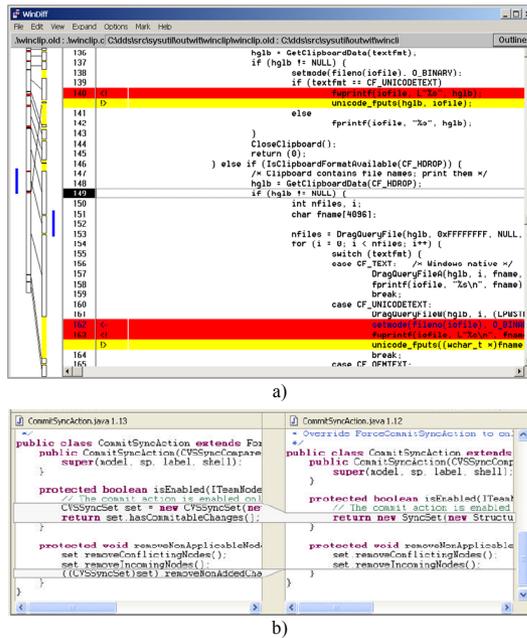


Figure 2.4: Visualizing changes between two versions of a file: (a) using WinDiff [124]; (b) using Hipikat [27].

of file versions, such tools cannot give an evolution overview of real-life projects that have thousands of files, each with hundreds of versions. Furthermore, they do not exploit the entire information potential of SCMs, such as information related to the time and author of changes between two versions.

More recent tools try to generalize this to evolution overviews of real-life projects whose evolution spans hundreds of versions. *Historian* [59], for instance, offers a simple visualization of CVS repositories at file level using the Gantt chart paradigm [50] (see Figure 2.5). This visualization, however, works well only on a small number of files and does not offer overviews of evolution for entire projects.

In a different approach, Collberg *et al.* visualize with *Gevol* [22] software structure and mechanism evolution as a sequence of graphs (see Figure 2.6). However, their approach does not seem to scale well on real-life data sets containing hundreds of versions of a system.

VRCS [67] and *3DSofVis* [93] try to improve the scalability issue by using time as a separate dimension in a 3D setup. While this approach allows the visualization of a larger number of versions, it suffers from the inherent occlusion problem of 3D visual environments, thus decreasing the overview capabilities of the visualization.

Lanza [69] uses the *Evolution Matrix* to visualize object-oriented software evolution at class level (see Figure 2.8). Closely related, Wu *et al.* [125] use the *Evolution Spectrograph* to visualize the evolution of entire projects at file level and visually emphasize the moments of evolution (see Figure 2.9). These methods scale very well with

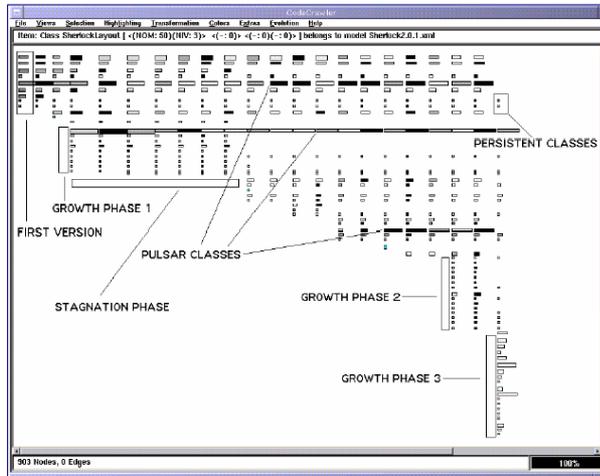


Figure 2.8: Visualization of software evolution at class level using the Evolution Matrix [69]. Time is encoded in the horizontal axis. Every rectangle depicts a class in the system. The width of each rectangle encodes the number of methods, height encodes the number of variables, color encodes size modification: black = increase, grey = decrease, white = constant.

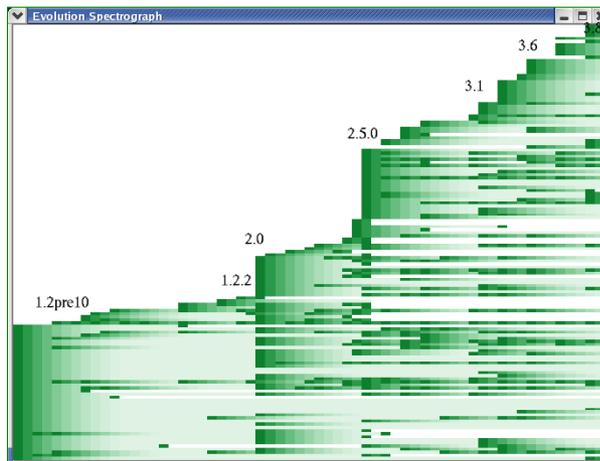


Figure 2.9: Visualization of software evolution at file level using the Evolution Spectrograph [125]. Time is encoded in the horizontal axis. Every horizontal line depicts a file. Color encodes the release of a new version of a file: green = new version, white = old version. As the time passes, versions become older and their color changes from green to white.

industry-size systems and provide comprehensive evolution overviews. Still, they do not offer an easy way to determine the classes and files that have a similar evolution. Furthermore, they address a relatively high granularity level and provide less insight into lower-level system changes, such as the many, minute source code edits done during debugging.

Not only the evolution of structure is important for software evolution analysis but also the evolution of quality metrics. These are particularly important for supporting the management decision process by detecting software quality trends. The tools presented above can visualize at most three quality metrics at once (*i.e.* the Evolution Matrix presented above visualizes number of methods, number of variables and size change status). Pinzger *et al.* [91] proposed with *RelVis* a novel method to visualize the evolution of a larger number of metrics using Kiviat diagrams (see Figure 2.10). They based their visualization on the release history database engine constructed by Fischer *et al.* [45], in an effort to provide an integrated framework for evolution data extraction, analysis, and visualization. However, their approach can only handle a small number of software versions.

One of the farthest-reaching attempts to unify all SCM activities in one coherent environment was proposed by German *et al.* with *SoftChange* [51]. Their initial goal was to create a framework to compare Open Source projects. Not only CVS was considered as data source, but also project mailing lists and bug report databases. *SoftChange* concentrates mainly on basic management and data analysis and provides simple chart-like visualizations (see Figure 2.11).

In another recent attempt, Burch *et al.* [15] proposed *EPOSee*, a framework for visualization of association and sequence rules extracted from software repositories using *eROSE* [131] as data mining tool (see Figure 2.12).

Concluding, a number of software evolution visualization tools have been proposed by the research community. The most important compromise they try to make is between revealing the structure of a software system and its evolution. These tools appear to be useful instruments for getting insight in the evolution of software. Nevertheless, many of the requirements presented in Section 2.4.1, for instance R1, R3, and R5 are little addressed or not at all. The scalability (R2) appears to be another important limitation of many tools either in terms of code size they can address, or in number of versions. Finally the proposed visualizations (R4) enable a limited number of evolution investigation scenarios, and their effectiveness needs to be more thoroughly evaluated. Relating these issues to the findings of Bassil and Keller [7] may explain the lack of acceptance and popularity of these software evolution visualization tools in the software engineering community.

2.5 Conclusions

In this chapter, we have given an overview of the place of software evolution visualization in the larger context of software engineering activities. We have introduced software evolution visualization as a component of the maintenance activities performed during the lifetime of a software project. Just as other visualization techniques, software evolution

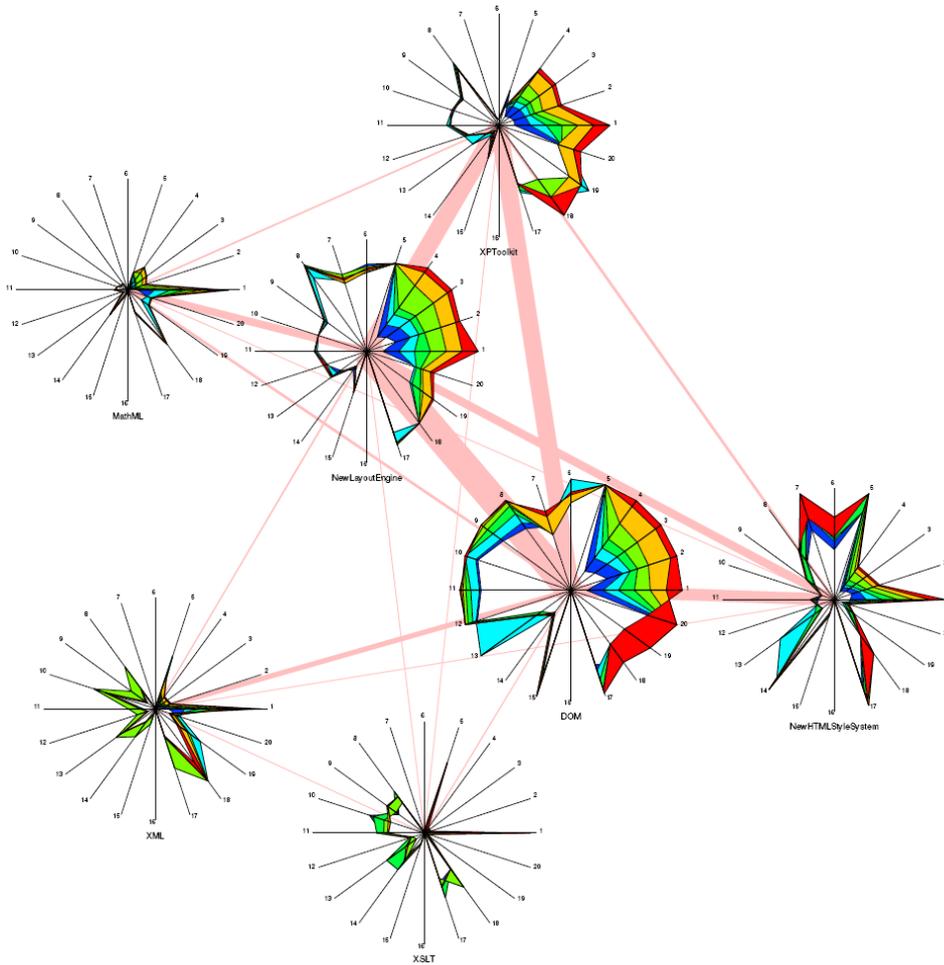


Figure 2.10: RelVis [91]: visualizing the evolution of 20 metrics along 7 releases for 7 software modules using Kiviats diagrams. Kiviats axes indicate metrics; color encodes releases; edge thickness encodes logical coupling between modules.

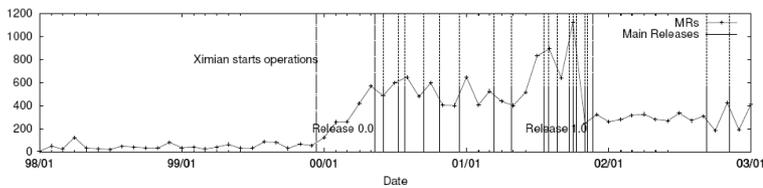


Figure 2.11: Visualization of software evolution in SoftChange [51]. The graphic shows the evolution in time of the number of modification requests.

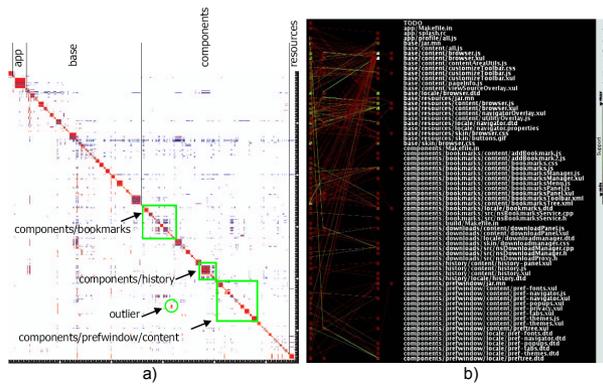


Figure 2.12: Visualization of evolution association rules between files with EPOSee [15]: (a) using a matrix representation; (b) using parallel coordinates.

visualization could be used not only to check a hypothesis on a given dataset, but also to discover the unexpected. Software evolution visualization is a natural complement to two data analysis techniques: the data analysis of the software evolution, which extracts facts and metrics concerning the evolution in time of a given software corpus, and the classical reverse engineering, which extracts facts and metrics concerning a single software version. Ideally, software evolution visualization should be integrated seamlessly with software configuration management (SCM) systems and various analysis and fact extraction tools to provide views on the evolution of a software system for a wide range of aspects.

In practice, we are still very far from the above ideal situation. Concluding our review, it appears that data management, evolution data analysis, and evolution visualization activities have little or no overlap in the same tool (Table 2.1). Reverse engineering tools are still an active area of research, and it is not simple to find reliable and scalable static analyzers and fact extractors for arbitrary code repositories. Evolution data analysis tools, being a newer research area, have still a long way to go to deliver insightful, unambiguous facts and metrics on the changes in a project. Given the relative novelty of such tools, coupled with the immaturity of data access APIs to code repositories, there are rather few visualization tools that target software evolution. These tools can be improved in many respects:

- the type and number of facts and metrics whose evolution is displayed;
- the scalability of the tools in presence of nowadays' huge software code bases;
- the intuitiveness of the visual metaphors chosen to display the extracted facts;
- the integration of visualization with software evolution data mining and analysis techniques;
- the validation of the proposed methods and techniques on real-world cases.

Making steps in the direction of a software evolution visualization toolset that satisfies these requirements is the focus of the next chapters of this thesis.

Chapter 3

Software Evolution Domain Analysis

In this chapter we present an analysis of the software evolution domain. We propose a generic description for system evolution, and we use this description to construct a formal model of software evolution. We also address here practical data management and analysis issues related to mapping available evolution information on this general model. Next, we use the constructed model to present and formalize the problem of software evolution. We also use this model in the remainder of this thesis as a backbone for several visualizations of software evolution.

3.1 Introduction

Software evolution analysis is a promising approach to facilitate system and process understanding in the maintenance stage of large software projects. Nevertheless, at this moment there are no tools that explicitly provide high-level information on the evolution of software. Software Configuration Management (SCM) systems introduced in the previous chapter explicitly record information on changes in software, albeit at an unstructured, text file level.

In the last decade, SCM systems have become an essential ingredient of efficiently managing large software projects [16], and therefore, they have been used to support many “legacy” systems (*i.e.*, large systems that evolve mainly by building on previously developed software). In practice, SCM systems are primarily meant for manually navigating the intermediate versions of a software system during its evolution. The information that such systems maintain is focused strictly on this purpose, *i.e.*, tell the user which file(s) have changed when, and who changed them, during the evolution of a set of files, which is called a repository. This functionality can be seen as providing a very limited view on software evolution at file granularity level. However, as discussed in the previous chapters, software maintenance requires answering more complex queries, which relate

to examining the evolution of software data at several other levels of detail than code files, and also examining the evolution of more quantities than just the source code, for instance software metrics (see [43, 65]).

Figure 3.1 summarizes the tasks of the software evolution analysis domain, including key activities and entities. A model for software evolution has a central position.

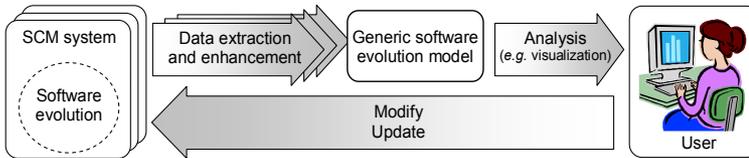


Figure 3.1: Software evolution visualization domain tasks. A generic software evolution model enables a standard visualization methodology for the analysis of evolution information from a large range of SCMs.

The goal of this chapter is to present a generic model of software evolution. A particular (simple) instance of this model is the evolution of software such as recorded by SCM systems. Other (more complex) instances are the evolution of software artifacts at various other granularity levels, such as functions or modules, or of non-structural artifacts, such as software metrics. The use of this model is to establish a common methodology for the several variants of evolution analysis which are encountered in the practice of software maintenance. Our model should be, for example, capable to abstract software evolution across programming language barriers and/or choice of the software metrics. A second aim of the proposed model is to support the implementation of more complex evolution analysis scenarios based on the elementary evolution data maintained by typical SCM systems. In this way, we can use the common subset of low-level information accessible in most SCM systems to construct generic, extendable analyses of software evolution as demanded by various application scenarios. Finally, we use the proposed software evolution model to construct a methodology for visual evolution analysis of software systems. Concrete applications of the model to several types of problems and software artifacts are described in the following chapters.

In the next section, we give a generic definition of the evolution of systems in general. In Section 3.3, we particularize this generic description to the evolution of software systems. An important step of this process is to detail the concept of *similarity* for software systems. We explain how we instantiate our generic evolution model using the concrete software evolution data available in practice. To this end we use data extracted from CVS [28] and Subversion [104] repositories, two of the most popular SCMs used in practice (Section 3.4). The challenges related to visualizing the proposed model for software evolution are presented in the Chapter 4 together with a standard methodology for addressing them.

3.2 System Evolution

In general terms, evolution refers to a process of change in a certain direction. As a consequence of evolution, systems can either increase or decrease in complexity. In software,

it is widely accepted that the complexity of systems only increases as they evolve in time (Lehman's second law of software evolution [70]). In the following, we describe system evolution with a bias towards software systems. We build the evolution description from the perspective of an external human observer interested in making judgements about the corresponding system.

A system at a particular moment can be described as a collection of entities:

$$S = \{e_i | i = 1, \dots, n_S \in \mathbb{N}\}.$$

An entity is usually characterized by a set of attributes:

$$A(e_i) = \{a_j | j = 1, \dots, n_A \in \mathbb{N}\}.$$

Each attribute has values of a certain type, in a certain domain $a_j \in D_j$. For example, a software system can consist of two files $S = \{F_1, F_2\}$. Each file has a number of attributes:

$$A(F_i) = \{\textit{name}, \textit{size}, \textit{type}, \textit{number of lines}, \textit{author}\},$$

where $\textit{name} \in \textit{Strings}$, $\textit{size} \in \mathbb{N}$, $\textit{type} \in \textit{Extensions list}$, $\textit{number of lines} \in \mathbb{N}$, $\textit{author} \in \textit{Team list}$.

A given system at a certain moment can be described in many different ways. Such descriptions can be structured as a hierarchy, where each level describes the system at some level of detail. This usually implies a containment relation between the entities at various levels. For example, the previous system S of two files can be described at a line level, if we assume every file F_i can be seen as an (ordered) collection of lines:

$$F_i = \{l_j | j = 1, \dots, n_{F_i} \in \mathbb{N}\}.$$

For a finer level of detail, every line can be considered to be a sequence of bytes:

$$l_j = \{b_k | k = 1, \dots, n_{l_j} \in \mathbb{N}\}.$$

Hierarchical descriptions are useful for two reasons. First, some information is inherently hierarchic, so it is best described in this way, such as the structure of a file system. Secondly, hierarchies can be generated when needed in order to simplify a given system and reduce the complexity of the analysis task.

We are interested in describing the evolution of software systems. Such systems do not have a continuous evolution. That is, their evolution can be seen as a set of discrete states in time: $S(t_1), S(t_2), \dots, S(t_n)$, where $t_i \in \mathbb{R}^+$. For simplicity we shall denote $S(t_i)$ (i.e., S at time t_i) by S^i , and an entity $e \in S^i$ by e^i .

To characterize system evolution, we hence have to look at the evolution in time of entities and attribute values over the sequence $\{S^i | i = 1, \dots, n_V \in \mathbb{N}\}$. When analyzing the evolution of discrete systems, one is often interested in answering questions such as "What has changed / stayed the same?", "How much was something changed?" or "What was created / disappeared?". To do this we must be able to relate S^i with S^j , where $i \neq j$. That means we must relate entities e^i with e^j , and potentially also corresponding entity attribute values. However, to be able to compare attributes values, we must first be able to relate and compare corresponding entities. So we focus first on entity comparison.

In order to compare entities, the generic notion *entity similarity* is introduced. Let S^i

be the set of entities describing the state of a system at time t_i , and S^j the set of entities describing the state of the same system at a later time t_j . *Entity similarity* (Γ_{ij}) gives a correspondence between the elements of $S^i = \{e_k^i | k = 1, \dots, n_{S^i} \in \mathbb{N}\}$ and those of $S^j = \{e_l^j | l = 1, \dots, n_{S^j} \in \mathbb{N}\}$, describing how entities in S^i relate to those in S^j . Formally, this may be defined as a mapping from the set of pairs (e_k^i, e_l^j) to the interval $[0, 1]$, describing the similarity ratio between the two entities of a pair.

Definition 3.2.1 *Entity similarity*

$$\Gamma_{ij} : S^i \times S^j \rightarrow [0, 1]$$

$$\Gamma_{ij}(e_k^i, e_l^j) = \begin{cases} 1 & | e_k^i \text{ evolved into } e_l^j \text{ with no modifications} \\ \text{const} & | e_k^i \text{ evolved into } e_l^j \text{ by modifications} \\ 0 & | e_k^i \text{ did not evolve into } e_l^j \end{cases}$$

where $\text{const} \in (0, 1)$ is a measure of the similarity of e_k^i and e_l^j . In a concrete application, const may be a function depending on entity specific structure and / or attributes. In other words, the following possibilities for relating e_k^i with e_l^j exist:

- they represent the same object (or clones thereof) to the external human observer, and their attribute values are identical;
- they represent the same object (or clones thereof) to the external observer, but a set of relevant attributes have different values;
- they represent unrelated objects to the external observer.

Instead of similarity, we can alternatively measure change, which is a complementary measure. Depending on the concrete application, it might be more convenient and/or intuitive to talk about change than similarity or vice versa.

Take for example a software system that consists initially of two files: F_1 and F_2 . The system can be described at the initial moment as $S^1 = \{F_1^1, F_2^1\}$. Suppose that in time a developer edits file F_1 deleting 20% of the lines, leaves F_2 alone, and creates a new file F_3 . After these events, the system can be described as $S^2 = \{F_1^2, F_2^2, F_3^2\}$. A possible entity similarity function Γ_{12} describing the correspondence between the entities of S^1 and those of S^2 in terms of Definition 3.2.1 could be:

$$\begin{aligned} \Gamma_{12}(F_1^1, F_1^2) &= 0.8 & \Gamma_{12}(F_1^1, F_2^2) &= 0 & \Gamma_{12}(F_1^1, F_3^2) &= 0 \\ \Gamma_{12}(F_2^1, F_1^2) &= 0 & \Gamma_{12}(F_2^1, F_2^2) &= 1 & \Gamma_{12}(F_2^1, F_3^2) &= 0 \end{aligned}$$

Note that the concrete expression of Γ can involve also attribute values (or functions thereof) of the considered entities. In the example above:

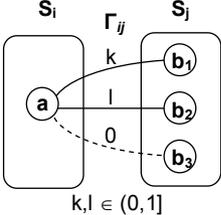
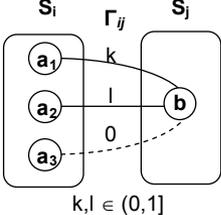
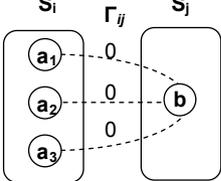
$$\Gamma_{12}(F_k^1, F_l^2) = \begin{cases} \frac{\min(|F_k^1|, |F_l^2|)}{\max(|F_k^1|, |F_l^2|)} & | F_k^1 \text{ and } F_l^2 \text{ address the same file} \\ 0 & | F_k^1 \text{ and } F_l^2 \text{ address different files} \end{cases}$$

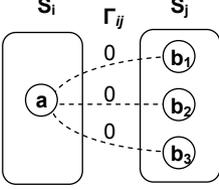
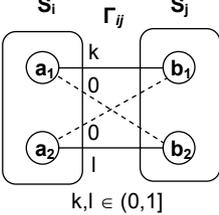
where $|F|$ denotes the size of a file F .

The entity similarity function Γ is not unique. Being given a system at two moments S^i and S^j there can be many functions of the type presented in Definition 3.2.1 that

describe how the corresponding entities relate. These functions give the perspective of different observers on system change. Therefore, they support an observer-centered view on system evolution.

The Γ type functions given by Definition 3.2.1 are useful because they can encode various changes in a system which are of interest to the external human observer. We identify five patterns of change among many other possible: *split*, *merge*, *insertion*, *deletion*, and *continuation*. These patterns are of particular interest for the assessment of software evolution. We detail them next.

<p>Definition 3.2.2 Split</p> <p>An entity $a \in S^i$ has been split into a set of entities $S_{split} \subseteq S^j$ during system evolution from S^i to S^j, if and only if:</p> $(\forall b \in S_{split} : \Gamma_S(a, b) \neq 0) \wedge$ $(\forall c \in S^j - S_{split} : \Gamma_S(a, c) = 0)$ <ul style="list-style-type: none"> • If $a \in S^i$ is split into a set of entities S_{split}, then S_{split} is unique. Let $S_{split}(a)$ denote the associated split set of a. 	 <p>Example: a split into $\{b_1, b_2\}$ $S_{split}(a) = \{b_1, b_2\}$</p>
<p>Definition 3.2.3 Merge</p> <p>An entity $b \in S^j$ has been merged from a set of entities $S_{merge} \subseteq S^i$ during system evolution from S^i to S^j, if and only if:</p> $(\forall a \in S_{merge} : \Gamma_S(a, b) \neq 0) \wedge,$ $(\forall c \in S^i - S_{merge} : \Gamma_S(c, b) = 0)$ <ul style="list-style-type: none"> • If $b \in S^j$ is merged from a set of entities S_{merge}, then S_{merge} is unique. Let $S_{merge}(b)$ denote the associated merge set of b. 	 <p>Example: b merged from $\{a_1, a_2\}$ $S_{merge}(b) = \{a_1, a_2\}$</p>
<p>Definition 3.2.4 Insertion</p> <p>An entity $b \in S^j$ has been inserted during system evolution from S^i to S^j, if and only if its associated merge set is empty, i.e. $S_{merge}(b) = \emptyset$.</p> <ul style="list-style-type: none"> • Let $S_{inserted} \subseteq S^j$ be the set of all entities inserted during system evolution from S^i to S^j. 	 <p>Example: b inserted $S_{inserted} = \{b\}$</p>

<p>Definition 3.2.5 <i>Deletion</i></p> <p>An entity $a \in S^i$ has been deleted during system evolution from S^i to S^j, if and only if its associated split set is empty, i.e. $S_{split}(a) = \emptyset$.</p> <ul style="list-style-type: none"> Let $S_{deleted} \subseteq S^i$ be the set of all entities deleted during system evolution from S^i to S^j. 	 <p>Example: a deleted $S_{deleted} = \{a\}$</p>
<p>Definition 3.2.6 <i>Continuation</i></p> <p>An entity $a \in S^i$ has been continued with $b \in S^j$ during system evolution from S^i to S^j, if and only if $S_{split}(a) = \{b\}$ and $S_{merge}(b) = \{a\}$.</p> <ul style="list-style-type: none"> Let $S_{continued} \subseteq S^i \times S^j$ be the set of all entities that are continued during system evolution from S^i to S^j, and their associated continuation elements. 	 <p>Example: $S_{continued} = \{(a_1, b_1), (a_2, b_2)\}$</p>

The set of above change patterns for a given system represents its evolution (Δ_Γ) as observed from the perspective of a given Γ (i.e., external observer). Although this definition is generic, there exist also other alternatives for defining Δ_Γ . For instance, definitions that focus on dynamic aspects can be imagined. However, the definition given above is particularly useful for characterizing the evolution of systems described by their contents, such as software systems. In this context, different Γ 's refer to different assessments of what and how much has changed. Hence, Γ has in practice to be chosen such that it captures change related information of interest for a given use-case. This is a problem and context dependent procedure.

In the remaining of this thesis we show concrete examples of insertion, deletion and continuation patterns during the evolution of software systems. We do not show split and merge patterns. Such evolution patterns are based on difficult to compute Γ functions in the context of available sources of data about software evolution. Computing these functions forms the subject of another direction of research, i.e., *software repository mining* (see [45, 130]) and, therefore, they are outside the focus of the work presented in this thesis.

Example: System evolution

Consider the situation depicted in Figure 3.2 which defines an entity similarity function between two states of a system S^i and S^j . Intuitively, Figure 3.2 describes the evolution of the system between the two states: one entity is deleted (a_3), one entity is inserted (b_3), and two other entities (a_1, a_2) are split and their results are merged to create two new entities (b_1, b_2). Using the previously introduced evolution patterns and Definition 3.2.1, one can summarize the evolution (Δ_Γ) as a tuple of the following sets:

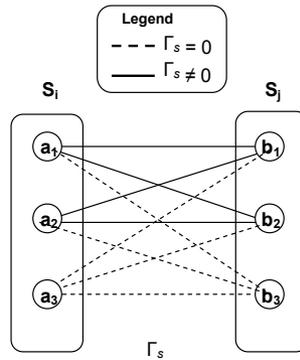


Figure 3.2: Example evolution patterns

- Deleted elements = $\{a_3\}$
- Inserted elements = $\{b_3\}$
- Continued elements = \emptyset
- Split elements and associated split sets = $\{(a_1, \{b_1, b_2\}), (a_2, \{b_1, b_2\})\}$
- Merged elements and corresponding merge sets = $\{(b_1, \{a_1, a_2\}), (b_2, \{a_1, a_2\})\}$

In order to answer the evolution questions previously identified, one has to investigate not only how entities relate in time but also how attributes thereof do. The Γ similarity functions allow entity correlations. Once a correspondence between system entities at different moments has been built, this can be used to relate also entity attribute values. In this thesis (see Chapters 5,6, and 7) we present a number of visualization techniques that can be used to make cross-correlations between the evolution of entities and that of entity attribute values.

3.3 Software Evolution

We would like now to particularize the system evolution model presented in Section 3.2 for software systems. This translates into identifying the possible valuations for entities e , attributes a , and similarity function Γ that make sense and are useful to understanding the evolution of software.

Software evolution focuses on the states that a software system has during its life time. It refers to the changes the system entities and their attributes undergo, from the moment the development starts, then throughout maintenance, until the system is terminated. The set of entities may include not only source code, but also supporting information, like design documents or project management plans. Additionally, the set of entities does not refer only to digital content. It may include, for example, document hardcopies and other material samples intended to support the development process (*e.g.*, yellow sticker notes on a board). This thesis addresses only the evolution of the source code. Hence, the term

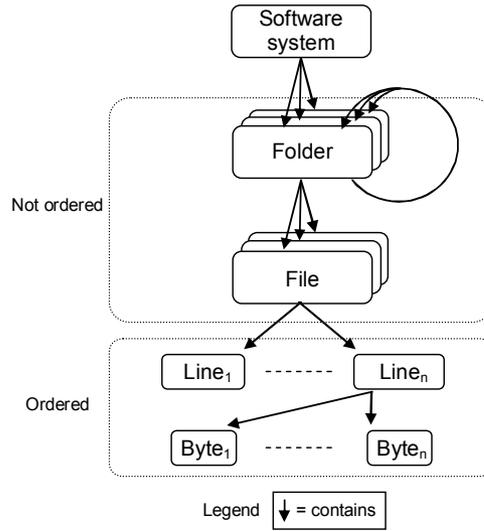


Figure 3.3: Software system model

software evolution will be further used to refer to the evolution of source code of software systems.

The stated goal of software evolution analysis is to support the system and development process understanding during the maintenance phase of a project. Software structure understanding is a main component of this, and is often used as foundation for further understanding the system behavior or the development process. The basic assumption for structure understanding is that developers are familiar with software representations that are similar to the ones they use to construct it [37]. Therefore, an efficient analysis should be based on a software model that emphasizes the building blocks and the main construction mechanisms. To this end we propose a hierarchical model of software generally accepted by the developers community. This model is shown in Figure 3.3.

Let S be a software system. At a high level of detail, the content of a S may be defined as a set of files arranged in a hierarchy of folders:

$$S = \{file_i | i = 1, \dots, n_S \in \mathbb{N}\}.$$

At a next granularity level, each file can be seen as an ordered collection of lines:

$$file_i = \{line_j | j = 1, \dots, n_{file_i} \in \mathbb{N}\}.$$

At the lowest level of detail, a line may be represented as an ordered set of bytes:

$$line_j = \{byte_k | k = 1, \dots, n_{line_j} \in \mathbb{N}\}.$$

The above model defines a possible hierarchical structure of a file as a set of lines formed by bytes. This hierarchy is useful, for example, when comparing or analyzing files on a line-by-line basis. Clearly, other alternative hierarchies exist. For example, we can define a file as a set of syntactic constructs, each consisting of an ordered set of tokens. Such a model would be useful for making higher level comparisons between files, for example,

in terms of a valid construct written in a given programming language.

To describe the evolution of a software system in terms of this hierarchical model using the definition of evolution Δ_Γ introduced in Section 3.2, entity similarity functions Γ have to be provided. For a complete characterization of evolution, such functions have to be defined at each level, that is, on each entity type: *files*, *lines*, and *bytes*. If such functions are defined at one level, then we can compare software at different moments at that level. If no similarity functions are defined at a hierarchy level l , one can in practice construct such functions using a function Γ' at a level l' lower than l . Concretely, if we know an inclusion/containment relation that maps an entity e on level l to a set of entities $\{e'_i | i = 1, \dots, n \in \mathbb{N}\}$ on l' , then we can use Γ' to construct a valid Γ function. However, the expressivity of this Γ function is limited to the knowledge encapsulated in the function Γ' , and in the containment and order relations between elements.

For example, let S be a software system made of two files F_1 and F_2 . Suppose file F_1 has two lines s, t and file F_2 has also two lines u, v . Suppose during development file F_1 is modified such that line s is deleted, and file F_2 remains unmodified. Let Γ_l be a known similarity function at line level between the two moments of development, given by the formula:

$$\Gamma_l(l^1, l^2) = \begin{cases} 1 & | l^2 \text{ is an unmodified version of } l^1 \text{ in the same file} \\ 0 & | \text{otherwise} \end{cases}$$

Suppose no similarity function Γ_f is defined at the file level. Then we can construct one, by using the containment relation and the values of the Γ_l function as follows:

$$\Gamma_f(f^1, f^2) = \frac{\sum_{l^1 \in f^1, l^2 \in f^2} \Gamma_l(l^1, l^2)}{|\{l^1 | l^1 \in f^1\}|}$$

Valuating the above formulae:

$$\Gamma_f(F_1^1, F_1^2) = \frac{\Gamma_l(s^1, t^2) + \Gamma_l(t^1, t^2)}{|\{s^1, t^1\}|} = \frac{0 + 1}{2} = 0.5$$

$$\Gamma_f(F_1^1, F_2^2) = \frac{\Gamma_l(s^1, u^2) + \Gamma_l(t^1, v^2)}{|\{s^1, t^1\}|} = \frac{0 + 0}{2} = 0$$

Similarly, $\Gamma_f(F_2^1, F_1^2) = 0$ and $\Gamma_f(F_2^1, F_2^2) = 1$. This gives an indication of the amount of change incurred by a file during evolution, and consequently can be used as a similarity function to relate files.

For the sake of simplicity we disregarded the order relation between lines in the example above. In the software practice, however, it is often the case that order should be taken into account when a similarity of files based on their lines is constructed [32]. Additionally, the Γ function presented above measures similarity based on file content and not on its semantics. It may be the case that the content of a file changes but the file offers the same set of features to the system it belongs to (*e.g.*, a different component implementing the same interface). A Γ function that is oblivious to content changes but reveals changes in the feature set can also be constructed, as long as a way to validate file features based on its content is provided. This is rather difficult to implement in practice and forms the field of a different direction of research: *Software Test Automation* [44, 110].

3.4 Software Repositories

We have described so far how to model the evolution of a system and how to particularize this for one common representation software. In practice, the set of entities e , attributes a and similarity functions Γ characterizing the evolution of a software system are not readily available. They have to be inferred from the information stored in SCM repositories, the only instruments available for recording the history of software. These repositories are described next.

During the last decades, SCM systems have become an essential ingredient of effectively managing large-scale software development projects [16]. Many SCM systems are available on the market, such as CVS [28], Subversion [104], Visual SourceSafe [114], RCS [54], CM Synergy [20], and ClearCase [19].

One of the most popular ones is the CVS system, which is available via the Open Source community. This system concentrates mainly on maintaining a history of the artifacts developed in a software project. Additionally, CVS has been the preferred choice for SCM support in many Open Source projects in the last decade, and consequently, many repositories covering long (*i.e.* 5-10 years) evolution periods are freely available for analysis. Therefore, CVS is used in this thesis as starting point to illustrate the concepts and challenges of software evolution analysis.

Another SCM system that becomes increasingly popular is Subversion, which is also available via the Open Source community. Subversion offers similar functionality based on similar concepts, but tries to improve on the shortcomings of CVS. However, Subversion is relatively young and, therefore, there are few repositories available that cover a long period of time.

However, SCMs do not support a very rich set of entities e , attributes a and similarity functions Γ . SCM repositories are primarily meant for navigating the intermediate versions that a software system has during evolution. The information they maintain for this purpose corresponds to a limited software evolution description. For example, CVS repositories define basic similarity functions only for files and lines. The challenge in this context is to extend and enhance the evolution representation offered by SCM repositories such that more complex and relevant results can be obtained by evolution analysis. Another challenge of using SCM systems in software evolution analysis is the data acquisition step. As SCMs are not built to support data analysis, they do not offer a rich API with machine-readable output. This issue has been outlined in Section 2.2.

Next, we detail the particularities of the CVS and Subversion repositories, and we emphasize on the challenges they pose to evolution data analysis.

3.4.1 CVS

CVS assumes the software model shown in Figure 3.4. It maintains a central archive of all intermediate versions that a file has during development. In CVS terms, these versions are sometimes also called *revisions*. Compared to Figure 3.3, the *line* is the lowest level of detail offered by CVS.

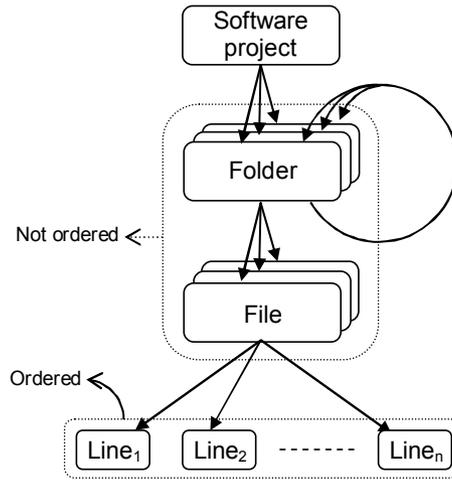


Figure 3.4: CVS model of software

Next, the inner mechanisms of CVS are described from the perspective of the entity similarity functions they provide or enable.

The unit of archiving in CVS is a file revision. File revisions contain and impose a storage order on lines. Revision names give an implicit file and folder hierarchy of the system. New revisions are stored in the archive during commit transactions. To record these transactions CVS uses a simple comparison of the full path of the files. That is, a file F^1 evolved into a file F^2 if the two files have the same name and path in the associated folder hierarchy. Additionally, if the operating system modification bit is set, the file is considered to be updated. Inserted, deleted, and split files are separately reported. This implicitly defines the following similarity function between the files of two consecutive versions S^a and S^{a+1} of a software project:

$$\Gamma_{file}(F^a, F^{a+1}) = \begin{cases} \text{same} & \left| \begin{array}{l} Path(F^a) = Path(F^{a+1}) \wedge \\ \text{update bit } F^{a+1} = 0 \end{array} \right. \\ \text{modified} & \left| \begin{array}{l} Path(F^a) = Path(F^{a+1}) \wedge \\ \text{update bit } F^{a+1} = 1 \end{array} \right. \\ \text{different} & \left| \text{otherwise} \right. \end{cases}$$

where $\text{same} = 1$, $\text{modified} \in (0, 1)$ is an arbitrary constant, and $\text{different} = 0$.

To enable parallel development on the same file CVS introduces the notion of *file branch*. This refers to the CVS supported possibility of splitting a file in two identical versions (*i.e.*, branches) that can be further independently developed. One of the two parts continues to be the main direction of development, while the second one is regarded as *experimenting* direction. When the modifications performed on the secondary branch are mature enough they can be integrated into the main branch by merging. However, this feature is not actively supported by CVS. Once created, a secondary development branch is not discontinued (removed) by a merging event. What happens after such an event is

that two branches remain, which refer to the same content in the main branch after the merging point.

To make better use of storage space, CVS does not store entire revisions of files, but merely differences in terms of lines between consecutive revisions. That is, CVS stores only the first revision of a file, and it encodes the following revisions as lines that have been deleted or inserted with respect to the previous revision. In this way a given revision can be reconstructed starting from the initial one, by recursively applying the patches that are stored for each revision up to the given one. If the same number of lines are deleted and then inserted at a given position in a file, they are considered to be *modified*. This approach implicitly defines a similarity function Γ_l at line level for all the lines in a file between every two consecutive revisions of a file F^a and F^{a+1} :

$$\Gamma_{line}(l^a, l^{a+1}) = \begin{cases} \text{same} & \left| \begin{array}{l} l^a = l^{a+1} \wedge \\ \text{position } l^a|_{F^a} = \text{position } l^{a+1}|_{F^{a+1}} \\ \text{(ignoring inserted and deleted lines)} \end{array} \right. \\ \text{modified} & \left| \begin{array}{l} l^a \neq l^{a+1} \wedge \\ \text{position } l^a|_{F^a} = \text{position } l^{a+1}|_{F^{a+1}} \\ \text{(ignoring inserted and deleted lines)} \end{array} \right. \\ \text{different} & \left| \text{otherwise} \right. \end{cases}$$

where $\text{same} = 1$, $\text{modified} \in (0, 1)$ is an arbitrary constant, and $\text{different} = 0$.

Similarity Function Issues

The similarity functions introduced above (*i.e.*, Γ_{file} and Γ_{line}) are the only form of support CVS offers for recording the evolution of software projects. There are no similarity functions offered for complex syntactic constructs, such as functions or classes. The reason for this is simple: CVS tries to be as content-neutral as possible. Although implementations for these relations can be deduced, for instance using the hierarchy based approach described in Section 3.3, deriving a meaningful Γ is a quite complicated task. We are not aware of any publicly available implementation of a tool based on CVS that provides such higher-level similarity functions.

Another issue with the similarity functions defined by CVS is the lack of expressiveness. For example, CVS does not recognize files that have been moved in the source tree as identical. Moving a file to another location is recorded by CVS as a deletion of the file and a creation of a new one in the new location, without an explicit connection between the two events. Additionally, CVS does not detect line swapping. This translates into files being always reported as modified when lines are swapped, while in practice their semantic meaning could be the same.

Furthermore, out of the patterns defined in Section 3.2 only insertion, deletion and continuation patterns can be observed with Γ_{file} and Γ_{line} , and merging and splitting are not. Γ_{file} can be used to observe file splitting for parallel development. However, file splitting for refactoring and merging patterns are not covered. Consequently, code *drifting* and *refactoring*, important aspects of software evolution assessment, cannot be

directly monitored with the mechanisms provided by CVS.

To improve the quality of the software evolution assessment, the research community has recently tried to enrich the expressiveness of the similarity functions supported by CVS, such that refactoring, splitting and merging patterns can be observed too. The most common approaches are probabilistic and based on source code analysis results (see [61, 55]). Their heuristic nature, however, makes their acceptance difficult in software engineering practice.

Extending Similarity Functions

The similarity functions implicitly defined by CVS (*i.e.* Γ_{file} and Γ_{Line}) are available only between consecutive versions of a software system. However, if they exist between each two consecutive versions, implicit variants can be constructed between any two arbitrary versions as follows.

Let $S^n \dots S^m$ be a list of $(m - n + 1)$ consecutive versions of a software system, sorted in increasing order of archiving (commit) times. Assume similarity functions Γ exist between any two consecutive versions from this list. Then, a similarity function between entities of S^n and S^m can be defined as:

$$\Gamma_{nm}(x, y) = \begin{cases} \text{same} & \left| \begin{array}{l} \exists E = \{e_k \in U^k \mid k = n, \dots, m\}, \\ (\forall u \in E \cap U^k, \forall v \in E \cap U^{k+1} : \\ \Gamma(u, v) = \text{same}) \Big|_{k=n}^{m-1} \end{array} \right. \\ \\ \text{modified} & \left| \begin{array}{l} \neg \exists E = \{e^k \in U^k \mid k = n, \dots, m\}, \\ (\forall u \in E \cap U^k, \forall v \in E \cap U^{k+1} : \\ \Gamma(u, v) = \text{same}) \Big|_{k=n}^{m-1} \\ \wedge \\ \exists F = \{f_k \in U^k \mid k = n, \dots, m\}, \\ (\forall u \in F \cap U^k, \forall v \in F \cap U^{k+1} : \\ \Gamma(u, v) \neq \text{different}) \Big|_{k=n}^{m-1} \end{array} \right. \\ \\ \text{different} & \left| \text{otherwise} \right. \end{cases}$$

where $\text{same} = 1$, $\text{modified} \in (0, 1)$ is an arbitrary constant, $\text{different} = 0$ and

$$U^k = \begin{cases} \{x\} & |k = n \\ S^k & |n < k < m \\ \{y\} & |k = m \end{cases}$$

In other words, $\Gamma_{nm}(x, y) = \text{same}$ when it is possible to build a sequence of elements, one for each state of the system between n and m , starting with x and ending with y , such that the existing Γ between every two consecutive elements has value same . Similarly, $\Gamma_{nm}(x, y) = \text{modified}$ when such a sequence cannot be built, but a less restrictive one, that only requires that Γ between every two consecutive elements has other value than different . Finally, $\Gamma_{nm}(x, y) = \text{different}$ when none of the above sequences can be built.

3.4.2 Subversion

Subversion is a relatively new Open Source SCM that has recently gained a lot of popularity. It appeared in 2000 and its stated goal is to be “a compelling replacement for CVS”. In this respect Subversion is built on the same concepts as CVS, but it tries to address some of its shortcomings. The most important differences with respect to CVS are:

- The lowest level of entity detail in Subversion is the byte and not the line as in CVS. In this respect Subversion assumes the software model shown in Figure 3.3. Conceptually, this signals a shift in focus from pure text (line-based) files as in CVS to general binary files. This enables more accurate similarity functions at the syntactic construct level, in which the amount of change can more easily be taken into account. For example, a line where only one character changed could have a different relevance for analysis than a line where all characters changed;
- The similarity function on files recognizes files and folders that are moved in the source tree during development. This enables high level refactoring assessments of the system;
- The Subversion communication protocol is entirely human and machine parseable, which simplifies the task of data extraction from repositories. That is not the case with CVS which is partly human and partly machine readable, which makes the automatic interpretation and classification of output intended for human parsing difficult and error-prone.

While Subversion offers more functionality than CVS, it has the same approach towards software evolution recording. Namely, the Subversion repositories are primarily meant for navigating the intermediate versions that a software system has during evolution. Consequently, the information they maintain for this purpose corresponds to a limited software evolution description.

3.5 Conclusions

In this chapter we have presented a generic evolution model for systems. This model views evolution as a discrete set $\{S^i | i = 1, \dots, n_S \in \mathbb{N}\}$ of system states. Each state S^i is described by a set of entities $\{e_k^i | k = 1, \dots, n_{S^i} \in \mathbb{N}\}$ with associated attributes $\{a_l^i | l = 1, \dots, n_A \in \mathbb{N}\}$, at different levels of detail. Evolution (Δ_F) is observed via an application-specific similarity function Γ , and consists of five sets of change patterns: split, merge, insertion, deletion and continuation.

Next, we have particularized this model for software systems using one of the commonly accepted representations of software: a hierarchy of files containing lines of source code that consist of byte sequences.

Software is in practice stored in Software Configuration Management (SCM) systems, which provide approximations of the above evolution model (*i.e.*, e, a, Γ) and software representation (*i.e.* $software = file/line/byte$). CVS is a widely used system that

records change at line level and offers a limited information access API. Subversion is a newer system that records change at byte level and offers a better basic information access API.

However useful, the strength of both CVS and Subversion, and for that matter all other SCM systems we are aware of, is in the same time a weakness from our software evolution analysis perspective. Since these systems are content-neutral, similarity functions that detect software-specific or programming language-specific changes must be provided explicitly atop of the basic functionality. Consequently, an important challenge of software evolution analysis based on e , a and Γ data from SCM repositories is to derive a set of entities e' , associated attributes a' , and similarity functions Γ' that characterize the software evolution with respect to some use-cases of interest. This challenge can be addressed via a combination of reverse engineering and SCM repository data analysis, which is, however, not the focus of this thesis.

Once e' , a' , and Γ' are constructed, the challenge is to map these elements on visual representations in order to better and easier get insight into the system evolution $\Delta_{\Gamma'}$. Our approach towards addressing this issue is the subject of the next four chapters. To this end we use only e' , a' and Γ' elements that are readily available in, or can be easily inferred from CVS and Subversion repositories.

Chapter 4

A Visualization Model for Software Evolution

In this chapter we present a model for building visual representations of software evolution. The model is described in terms of the classical visualization pipeline, with five main elements: data acquisition, data filtering and enhancement, layout, mapping, and rendering. The model follows a generic approach, in which the flexibility of the pipeline elements is traded off against ease of use and efficiency in implementation. User interaction is an important part of the proposed model, as interactive exploration is at the core of the visual analysis process for understanding software evolution.

4.1 Introduction

In this chapter we describe the visualization model we use to represent the evolution data model introduced in Chapter 3 in a visual form. By a visualization model, we mean the set of design elements concerning the various aspects involved in a visualization, such as data selection and preprocessing, visual representation, and user interaction, which are combined to construct an application. The effectiveness of a visualization application is strongly influenced by decisions taken in the design of these various aspects of the visualization process, ranging from the choices of the graphics elements used to represent abstract data and the user interaction metaphors proposed to navigate the data space to the choices concerning application interoperability in a given methodology or process [17, 98].

Figure 4.1 shows the pipeline we propose to describe a large class of software evolution visualization applications. These applications, which target the various questions related to software evolution outlined in Chapters 1 and 2, are constructed to support the software evolution domain model presented in Chapter 3. Concrete applications are described in Chapter 5, 6, and 7, which present software evolution visualization at line, file and respectively system level. In the remainder of this chapter, we detail the com-

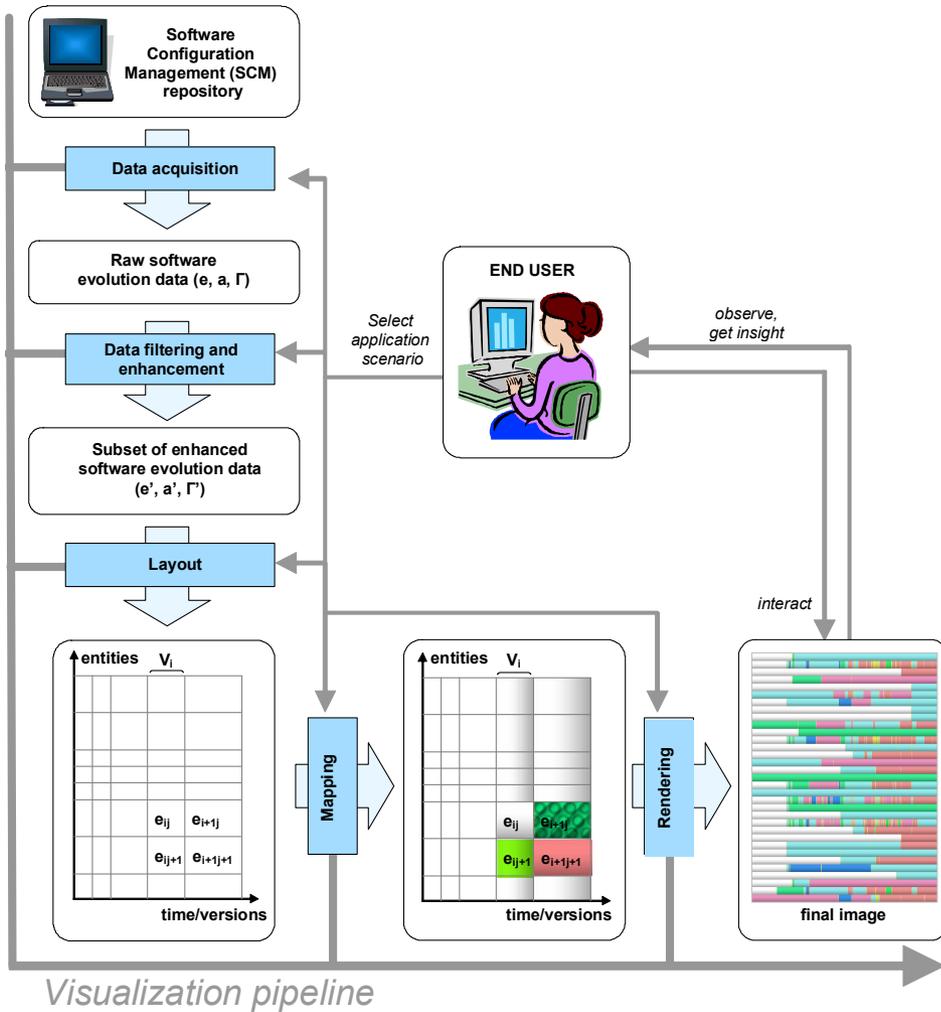


Figure 4.1: Visualization model for software evolution

mon design elements that are at the basis of these applications. We believe that these generic elements are useful in the design of a large class of visualization applications for time-dependent data, potentially going beyond software itself.

In Section 4.2, we detail the structure of the software visualization pipeline and comment on its similarities and differences with the classical visualization pipeline. Next, we present the several steps of the visualization pipeline. Section 4.3 describes the data acquisition from software repositories. Section 4.4 describes the data enhancement and filtering operations that enrich the basic repository information and help creating selections thereof. Section 4.5 describes the data layout, *i.e.*, the mapping of abstract data items to geometric shapes. Section 4.6 describes the construction of visible objects from the laid out shapes. Section 4.7 describes the final step in the visualization pipeline, *i.e.*, the rendering of the visible objects that produces the picture. We dedicated a separate section (4.8) to user interaction, as a careful design thereof has proved to be essential in the acceptance and success of software visualization applications. Finally, in Section 4.9 conclusions are drawn.

4.2 Software Visualization Pipeline

Our visualization model is similar to the classical visualization pipeline [17] (see Figure 4.2). It consists of a sequence of operations, which manipulate the data at hand, and datasets, which are concrete containers storing the software data. However, the evolution software visualization pipeline exhibits some particularities, which deserve special attention. These aspects are discussed next.

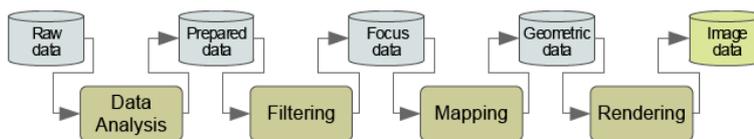


Figure 4.2: Classical visualization pipeline

The software evolution visualization pipeline starts with a *data acquisition* step (see Figure 4.1). At this step, the evolution information stored on Software Configuration Management (SCM) repositories such as CVS and Subversion is retrieved. The output of the data acquisition step is a basic software data model (Chapter 3) containing the raw information stored in SCM systems. Data acquisition is followed by *data filtering and enhancement*. At this stage the raw evolution data extracted from repositories is pre-processed and analyzed using, for example, data mining or software reverse engineering techniques. Additionally, subsets of interest of data and analysis results are selected for visual representation based, for example, on data properties. The next step in the visualization pipeline is *layout*, which assigns geometric position, dimension and shape to the entities selected for visualization. This step is not typically present in a scientific visualization pipeline, as scientific data often has an intrinsic layout, or spatial placement. Yet, the choices made in the data layout step are crucial in determining a good quality, easy to understand, scalable, and thus ultimately effective visualization. The layout operation

is followed by a closely related step, *mapping*, which specifies how data attributes map to visual attributes, such as color, shading, texture, transparency, and behavior of visual objects. *Rendering* is the last step of the visualization pipeline. In this step all entities selected for visualization are drawn using the position and appearance properties determined by the previous steps. After an image is rendered, users can analyze it, interact with it, and modify it by adjusting the parameters of the elements in the visualization pipeline. Consequently, *user interaction* closes a control feedback loop in the visualization model we propose.

Clearly, there are numerous choices involved in the design of each of the above steps. We can give a few examples. The time-dependent aspect of software can be mapped to a spatial axis, as in the case of a function plot, or can be shown via animation. For a given data layout, a data attribute can be mapped to a spatial dimension not used by that layout, *e.g.*, the third dimension for a 2D layout, or to color. Given the extent of this work, it was not possible nor practical to explore all combinations of choices. However, with our approach we tried to offer a generic visualization model which suitably covers a large class of visualization applications in our software evolution understanding domain. To this end we had to make several assumptions about the data and tasks at hand, and also to restrict the design choices in several directions. The resulting model is the focus of this chapter.

The model itself has been the incremental outcome of the construction and testing of several visualization applications that target specific data and tasks, which are presented in the following chapters. The specific choices of our visualization model are reflected into the structure of the visualization pipeline for software evolution which we discuss in the next sections.

The visualization model for software evolution understanding serves several purposes. First, it makes explicit which are the design invariants in our concrete visualization applications, and why we have chosen these. Choosing a limited set of design elements for a visualization application certainly limits the expression freedom in some respects, but also has the advantage of making the look-and-feel of several of such applications easier to follow and learn. Secondly, it makes the limitations of several design choices explicit. Thirdly, it serves as a guideline for the design of new visualization applications that target related tasks in the field of software evolution understanding, but also beyond.

We detail next the design decisions, invariants, and implementation of our visualization pipeline elements and explain them in the light of the requirements for software evolution analysis tools presented in Section 2.4.1.

4.3 Data Acquisition

The *data acquisition* step that commences the visualization pipeline extracts the evolution information from SCM repositories and makes it available for analysis. Given the nature, purpose and implementation of current SCM systems, this can be an error prone and resource consuming process (see Section 3.4). Additionally, the software evolution models differ slightly across repositories (see Sections 3.4.1, 3.4.2).

To cope with these issues we use specialized data extractors for each repository type, and a flexible database implementation to store data. As explained in Section 3.2, a software system at a particular moment in time can be seen as a set of entities $\{e_i | i = 1, \dots, n_S \in \mathbb{N}\}$ each entity being characterized by a set of attributes values $\{a_j | j = 1, \dots, n_e \in \mathbb{N}\}$. In the data acquisition step, we use a number of SCM specific data extractors that connect to different types of repositories over the network, retrieve the evolution data (*i.e.*, e , a and Γ), and store it in a generic database format. Essentially, this format can be seen as a set of database tables which store tuples of the type $(e_i, a_1, \dots, a_j, \dots)$, where the entity identity can serve as primary key. Such entity tables (T_{entity}) hold a single version of a given system, *i.e.*, there are as many entity tables as versions in the repository. Evolution similarity functions between two consecutive versions V^k and V^{k+1} of a system are also stored in tables as tuples of the type $(e_i, e_j, \Gamma(e_i, e_j))$ where entity $e_i \in V^k$ and entity $e_j \in V^{k+1}$. These tables are called similarity tables ($T_{similarity}$). An efficient design of the database schema and the actual database implementation is important in practice, as a single software repository can easily hold tens of gigabytes of data (see requirement R2 in Section 2.4.1).

We advocate to use an incremental, demand-driven database. Data table instances are created on-the-fly, as the user retrieves the corresponding software entity versions. Moreover, in the filtering stage discussed next in Section 4.4, the raw data is enriched with additional information obtained from specific data mining and analysis operations. This extra information can be well accommodated by supplementary data tables created on demand, which refer to the raw data tables. Most existing database engines on the market nowadays are able to implement the above model, so the actual choice can be made given other types of constraints, such as availability and overall system and platform integration. In our applications presented in Chapters 5, 6, and 7, we have used a SQLite [99] database for the Microsoft Windows operating system.

A second choice we made is to store data on the same workstation as where the filtering, layout, mapping and rendering steps are performed. In practice, this is almost always a different workstation than the one on which the actual software repository is located. This approach minimizes latency times between data selection and actual image rendering, as the large amount of data involved does not have to be sent always over the network, but only once during data acquisition. Additionally, we chose to populate the database on demand: data is brought from the remote repository, or locally created via filtering and enhancement, only when needed by a corresponding visualization task. This process is basically identical to the so-called *on-demand pipeline update* which traverses the visualization pipeline from the end (rendering) to the beginning (data acquisition) whenever the rendering requests a certain bit of information [118]. This is the pipeline execution model of choice in most modern visualization systems, as it creates (or updates) data only when needed, so it minimizes data storage and computation overheads. This yields very good results even for industry-size projects with hundreds of versions of thousands of files.

4.4 Data Filtering and Enhancement

Once the evolution information has been extracted from SCM repositories, it is preprocessed during the *data filtering and enhancement* step of the visualization pipeline. It is

at this stage that the limited, raw evolution information stored on repositories is enhanced and extended, by various analysis procedures such as data mining.

Several types of data enrichment operations can take place at the filtering stage. In the following, we give some examples of the types of operations that we have considered in the actual visualization applications described in the following chapters. The filtering and enhancement operations are presented in the following, starting with simple ones and ending with those that require a more complex implementation.

4.4.1 Selection

Data *selection*, also called querying, is the filtering process in which a subset of interest is constructed from a larger dataset. Different types of selection are common in the software evolution visualization process. The simplest selection picks a number of entities $e_i \in D$ of interest from a given input dataset D , based on entity identity or on the attributes a_j of these entities. Such selection operations are typical at the beginning of the insight-forming process, when the users limit their attention to some subset of interest. For example, in a source code repository, typical selection operations are to separate source code files from the other files, when one is interested in examining the code itself. Another typical selection is to separate the header files from the source code files, when one is interested in examining the evolution of the interfaces of a software system. These selections operate essentially as a filter on the *type* attribute of the file entities. Another common selection operation is to select the entities corresponding to a given attribute value range. For example, in large repositories containing the evolution of industry-size projects, it is common to select only a folder containing a subsystem, or alternatively the entity versions corresponding to a given time interval for further inspection. These selections operate as range value filters on the *path*, respectively *commit time* attributes of the file entities. A final example in the same category is to select the contributions of a given developer to a software system evolution, which amounts to a filter on the *author* attribute of the file entities. Examples of these selection operations are given in Chapter 5 and 6.

As explained, the result of a selection operation is a subset of entities. This subset can be stored as a data entity in the data model, separate from the actual software data entities, called a *selection*. This model has been used by several information visualization and software visualization frameworks, such as SoftVision [105], Tulip [4], Rigi [108], and GVF [76]. An alternative model is to create a new data attribute for every existing selection, and set the corresponding value to `true` for every entity belonging to that selection. This model is more similar to the so-called field dataset approach taken by scientific visualization frameworks such as ITK [62] and VTK [118]. The first approach is more memory efficient and also allows quick iteration over a selection's content, whereas the second allows a fast entity-in-selection query, and also provides a more uniform data model which simplifies the design of complex data filtering pipelines. As there is no clear winner, we have chosen to support both approaches in practice.

During the testing and use of our visualization applications, we found that a very wide range of queries on software evolution can be answered using a cascade of simple value-based selection queries.

4.4.2 Metrics

A second class of operations consists of the computation of *metrics*. Metrics are attributes that capture some quality aspects of the entities of the system at consideration. Metric computations are data enhancement operations that do not modify the value of existing entity attributes but create new ones. A typical use of such operations is to support several of the software metrics used in the reverse engineering practice [65, 43] (see requirement R3 in Section 2.4.1). In our context of software evolution understanding, metrics can be classified in three basic categories:

- *entity metrics*: These are metrics that are computed typically independently for every version of some given entity. All classical software engineering metrics fall in this category, as there is essentially no time, or evolutionary, aspect involved. Examples of such metrics are the number of lines of code or comments of a given entity, the number of functions or classes in a file, the cohesion or coupling between given software components such as classes, functions or packages, or the fan-in or fan-out of functional components. By computing version metrics and visualizing their evolution in time, for instance, using time series visualization techniques, trends in a software system evolution can be monitored (Chapter 6 and 7).
- *evolution metrics*: These are metrics that deliver a single value for all versions of a given entity. The purpose of such metrics is to characterize the entire evolution of a given entity (or set of entities) in a global manner. In contrast to the first class, these metrics are strongly related to the context of software evolution. Examples of such metrics are the average, maximum, and minimum number of lines of code a certain software entity (*e.g.*, file) has during its evolution, or the maximum number of authors who have modified such an entity during its entire lifetime. These metrics can be used for example to compare the evolution of several distinct entities from a global perspective. A more interesting example of an evolution metric is the *evolution similarity metric*, which compares how similar two files have evolved during their entire lifetime. This metric is presented in Chapter 6 where its use for obtaining and visualizing a system decomposition from the perspective of evolutionary-related components is discussed.
- *version metrics*: These are metrics that deliver a single value for all entities of some given type that belong to the same version of the system evolution. The purpose of version metrics is to globally characterize an entire system version. Examples of such metrics are the total number of files, functions, or lines of code in a given system version, or the number of bug reports or bug fixes in a given release. Such metrics are useful, for instance, for system architects to assess what the global quality is of a given software system release, and consequently decide if that release is ready for external dissemination or not.

All above mentioned metrics can be applied also on selections. Evolution metrics can be computed on a selected number of versions. Similarly, version metrics can be computed on a selected number on entities.

An illustrative way to visualize the three basic types of metrics mentioned above is to think of a software evolution dataset as a two-dimensional matrix where the lines cor-

respond to the different entities (*e.g.*, files) in a project and the columns to the different versions of each entity (see Figure 4.3). In this model, entity metrics correspond to, and must be visualized upon, the individual matrix cells; evolution metrics correspond to single values for each row; version metrics yield single values for each column. This metaphor is illustrative for the types of metrics and the involved visualization problems that emerge, as the actual layout which we use for depicting the software evolution is strongly related to the matrix image shown here (see Section 4.5).

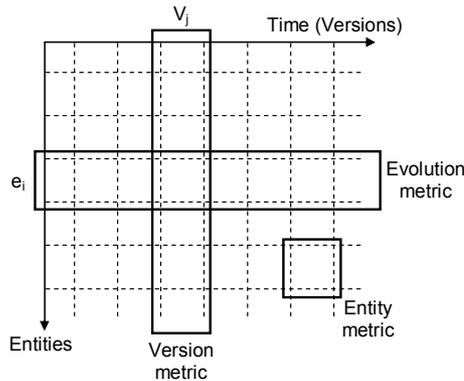


Figure 4.3: Basic metric categories in software evolution visualization

Composite metrics are hybrids between the basic entity, evolution, and version metrics. They deliver a single value for a specific selection which contains an application-dependent or analysis scenario-dependent set of elements. In most cases, composite metrics are just sums of basic metrics calculated on all entities in a given selection. However, depending on the actual selection semantics, more complex composite metrics can be designed. For example, if a selection contains all files involved in some software component, a specific composite metric can consist of a function of different types of evaluations done on the different types of files in the selection. For instance, a *sanity* metric can involve checking the presence and size of the documentation files, the number of comments in header files per function and class declaration, and the number of bug reports per line of code for the implementation files.

4.4.3 Clustering

As explained in Chapter 3, a software system admits many alternative structurings which reflect many different complementary aspects. For example, a software project can be hierarchically structured as a file system, component network, class or activity diagram, dependency graph, or developer responsibility graph. As software systems evolve, their structure also changes, so it is interesting to visualize these changes.

To this end *clustering* operations can be used. These data enhancement operations have several uses. First, they can emphasize the existence of similarities in a dataset given a certain perspective, encoded in the clustering criterion that drives the clustering process (see requirement R3 in Section 2.4.1). Second, clustering is useful as a generic

dataset size reduction operation. When there is too much data to examine (visually or otherwise), clustering can be used to reduce the subset of interest to a manageable size by grouping strongly related entities and treating them as atomic entities. Many of the classical visualization techniques of level-of-detail and context-and-focus are based on an underlying clustering operation that delivers a hierarchical system representation (see requirement R1 in Section 2.4.1).

Whereas selections group entities, metrics create data attributes for entities, clustering can be seen as creating new entities. These entities, or clusters, contain other entities, usually in a recursive, tree-like fashion. A clustering tree describes thus a system decomposition from a given perspective.

Providing clustering is important because it gives insight to the user, and SCMs offer little hierarchical structuring information. Concretely, the only hierarchical structuring information that such systems are guaranteed to offer is that of files-in-directories. This information is supported by SCM systems, as it is needed by the basic repository management tasks (check-in, check-out) that these systems have to provide.

Clustering operations can be grouped into two types:

- *version-based methods*: These methods cluster, or structure, a single version at a time. In this class fall the methods mentioned in Section 2.3 that recover system structure and architecture from its source code, for example, via parsing and code pattern matching. These methods are further studied in the field of static program analysis in reverse engineering, and are outside the scope of this thesis.
- *evolution-based methods*: These methods cluster, or structure, the evolution of entities. For example, entities whose evolutions are characterized to be similar can be grouped together in order to obtain a hierarchical system decomposition from an evolutionary perspective. We shall demonstrate this type of clustering operation in the context of file-level software visualization in Chapter 6.

4.5 Data Layout

Layout assigns a geometric position, dimension and shape to every entity to be visualized. We choose upfront for a 2D layout. Our need to display many attributes together may advocate a 3D layout. Yet, previous attempts to have 3D visualizations accepted by software engineers proved to be problematic [105, 75]. A 2D layout delivers a simple and fast user interface, no occlusion and viewpoint choice problems, and a result perceived as simple by software engineers. This design alternative has been previously advocated by the research community [121].

In particular, we opted for a simple 2D orthogonal layout that maps time or version number to the horizontal axis and entities e (e.g., lines, files, systems) to the vertical axis (Figure 4.1). Finally, entries are shaped as rectangles colored by the mapping operation (see Section 4.6). Other alternatives can be imagined, for instance, using both dimensions to describe entities and then arranging entity descriptions along a time axis. However, the layout we chose is simple to understand and therefore easier to accept by

the software engineering community. Additionally it is relatively easy to implement and scales very well with large amounts of data (see requirement R2 in Section 2.4.1). Within this model, several choices exist:

- *scale*: which level of detail for the entities from the repository should we visualize?
- *x-sampling*: how to sample the horizontal (time) axis?
- *y-layout*: how to order entities e on the vertical axis?
- *size*: how to size the rows and columns of the layout?

These choices are explained next.

Scale allows us to control at which level of detail we see the software repository. We have designed several so-called *views*, matching closely the most important entities of the generic software model presented in Section 3.3 (see requirement R1 in Section 2.4.1):

- the code view + the file view for visualizing repositories at source code line level, *i.e.*, $e = \text{line}$ (see Chapter 5);
- the project view for visualizing repositories at the file level, *i.e.*, $e = \text{file}$ (see Chapter 6);
- the system view for visualizing repositories at system level, *i.e.*, $e = \text{system}$ (see Chapter 7).

The horizontal axis can be *time* or *version* sampled. Time sampling yields vertical version stripes (V_i in Figure 4.1) with different widths depending on the length of the periods between commit times. This layout is good for project-wide overviews as it separates frequent-change periods (high activity) from stable ones (low activity). However, quick changes may result in overcrowded areas in visualization. The project view (Chapter 6) can be set to use this layout. Version sampling uses equal widths for all version stripes. This is more effective for entities that have many common change moments, for example, lines belonging to the same file. The file view (Chapter 5) uses this strategy by default.

The vertical axis shows entities e in the same version V_i . Two degrees of freedom exist here. First, we can choose in which order to lay out the entities e for a version. Secondly, we can stack the entities one above each other or use vertical empty space between entities. Both choices are detailed in Chapter 5.

4.6 Data Mapping

Mapping specifies how entity attributes, such as author, date, or type map to an entity's color, shading, and texture. As for layouts, concrete mappings are highly task-dependent and are discussed in the next chapters. Yet, we have identified several design decisions that reoccurred in all our visualizations:

- *Categorical* attributes, such as authors, file types, or search keywords are best shown using a fixed set of around 10 - 15 perceptually different colors. If more values need to be shown, for instance, in a project with 40 authors, colors are cycled. Categorical sets with less than 4.6 values can also be effectively mapped to carefully chosen simple texture patterns if the dimensions of the rectangles are above 20 pixels. Texture and color can be used to encode up to three independent attributes simultaneously.
- *Numeric* attributes, such as file size or age, bug criticality, or change amount, are most accurately shown using charts. However, when a color encoding is to be used, visually monotonous colormaps work best. We tried several colormaps: rainbow, saturation (gray-to-some-color), and three-color (e.g., blue-white-red). Interestingly, in most cases the rainbow colormap was the quickest to learn and accept by most software engineers and also by non-expert (e.g., student) users.
- *Structure* can be shown using shading. We use shaded parabolic and plateau cushions to show entities on different scales: file versions in file view (Chapter 5), files in project view (Chapter 6), and even whole systems in the system view (Chapter 7).
- *Sampling* is essential for overview visualizations. These can easily contain thousands of entities (e.g., files in a project or lines in a file), so more than one entity per pixel must be shown on the vertical axis for a complete overview. This can be addressed via antialiasing. We used this approach to render several entries per pixel line with an opacity controlled by the amount of fractional pixel coverage of every entry. This ensures a smooth visualization that is free of distractive artifacts, yet enables detection of outliers during a detailed inspection of the image. For a detailed description of this approach see Section 5.3.3. When outlier detection is the main analysis objective, a different approach can be followed. No color or transparency blending is performed. Yet, a non-restrictive dimension of the entity representation can be used to encode the presence and characteristics of the outliers. For a more detailed description of this approach see Section 7.3.2.

4.7 Rendering

After all position and appearance attributes have been set, in the rendering phase the final image is produced by drawing the constructed geometries with the chosen materials, lighting and view parameters.

The choice of a 2D orthogonal layout simplifies also the rendering step of the visualization pipeline. In this respect only zoom and pan operations are required to support user interaction. The 2D orthogonal layout requires also no real implementations of shading operations. All shading-like effects can be easily implemented as pre-computed textures, which speeds up the drawing process.

For rendering the graphic primitives we chose the OpenGL standard graphics interface. This facilitates the use of textures and blending operations, which we extensively use to render the encoding of entity attributes. Additionally, most OpenGL implementa-

tions make use of the hardware acceleration features available on workstations, speeding up considerably the rendering process.

4.8 User Interaction

User interaction is essential to our visualization model. It forms the feedback control loop of the visualization pipeline, as it enables users to adjust the parameters of all pipeline steps based on the analysis of the previously generated images. Consequently, it helps users to steer the process of building a mental model of data (see requirements R4 and R5 in Section 2.4.1). We provide a set of interaction techniques from the perspective of the information seeking mantra proposed by Shneiderman [96]: “overview first, zoom and filter, then details-on-demand”.

All layouts we propose (*i.e.*, the file, project and system views) offer comprehensive overviews of software evolution at different scales. To get detailed insight, zoom and pan facilities are provided. Zooming has a context dependent behavior – text annotations are shown only below a specific zoom level, whereas above another level antialiasing is enabled. We also offer two preset zoom levels to speed-up choosing commonly used view configurations: global overview (fit all software representation to window size) and one entity-per-pixel-line level.

Filtering is an important step of the visualization pipeline. In practice, this step is often performed manually according to criteria generated on-the-fly by visual inspection of data. Additionally, data filtering is a recurring activity throughout the entire visual assessment of software evolution. Consequently, a tight integration of this with the user interaction functionality is a must. We provide in all visualization mouse-based mechanisms for performing filtering. These mechanisms allow either individual or set picking of selections (*e.g.*, select individual files or select all files in a list between two given ones) and they are either entity or attribute based (*e.g.*, select files indicated by mouse clicks or select files created by author “x”). Most selections work on a range basis, *i.e.*, via user interface widgets that allow selecting a set of attribute value ranges from the domain of a given attribute.

To offer users direct access to detailed information about the entities depicted in an image we implement *details-on-demand* mechanisms in all visualizations. These mechanisms are based on a multiple views approach, in which details are displayed separately from the overview. The user indicates what entity to inspect in detail (*i.e.*, in the detail window) by pointing with the mouse to that entity in the overview window.

Using classical user interface techniques such as widgets and mouse-based brushing covers a wide range of the needed functionality, but does not cover the entire spectrum of possibilities. We have used also a number of less standard interaction widgets, such as the preset controller (Figure 6.9) and the cluster map (Figure 6.12). The use of these widgets is detailed further in Section 6.3.3 and 6.3.4.

4.9 Conclusions

In this chapter we have described the visualization model that we use to build graphical representations of software evolution, according to the evolution data model described in Section 3.4. Our goal is to build visualizations that are simple to follow, have a uniform look and feel, and allow an efficient and scalable implementation.

To this end, we offer a visualization model that follows closely the standard visualization pipeline discussed in the visualization literature (see [17]), consisting of the main steps of data acquisition, data filtering and enhancement, data layout and mapping, and data rendering. Yet, we made a number of decisions on various aspects of the pipeline used for our software evolution visualizations:

- We chose to store data locally in an incremental, demand-driven database to minimize access times, storage and computation overheads;
- We designed three types of filtering and enhancement operations: selection, metric computation and clustering. Selection operations are used for data filtering based on data identity, attribute type or values. Metric computations are data enhancement operations used to enrich entity attribute sets with software quality indicators. Clustering operations enhance entity sets with higher level descriptions of a software system.
- We opted for a 2D orthogonal layout that is simple to understand by the target users, and enables computationally efficient and scalable implementations;
- We use color to encode both categorical and numeric attributes, shaded cushions to encode structure, and antialiasing to obtain smooth visualizations free of distracting artifacts;
- We use the OpenGL standard graphics interface to take advantage of the hardware acceleration features available on most workstations.

We favor the use of a uniform design for the visualizations of different types of data. For example, we reuse the simple 2D orthogonal axis-aligned layout to show the evolution of several entities, such as lines of code versus file versions (Chapter 5) or source code files versus project releases (Chapter 6), but also non-evolution information such as project clusters versus level of detail (Chapter 6). This minimizes the effort required to switch the context when passing from one visualization to another.

User interaction is an important part of our model and implements a feedback control loop on the visualization pipeline. When implementing it in concrete applications we use many preset configurations and only a few user configurable parameters.

In the next chapters we instantiate the visualization model that we propose for a number of software entities commonly used by the software engineering community and with readily available evolution information from CVS and Subversion repositories. In Chapter 5 we visualize the evolution of lines of code, in Chapter 6 the evolution of files, and in Chapter 7 the evolution of software systems.

Chapter 5

Visualizing Software Evolution at Line Level

In this chapter we investigate how developers can be enabled to get detailed insight in the history of individual files. The aim is to enable them to understand the file status and structure better, as well as the roles played by various contributors. To this end, we propose an integrated multiview environment. Central to this visualization is a line-oriented display of the changing code, where each file revision is represented by a column, and where the horizontal direction is used for time. Separate linked displays show various metrics, as well as the source code itself. A large variety of options is provided to visualize a number of different aspects. Informal user studies that we have performed demonstrate the effectiveness and efficiency of this approach for real world use cases.

5.1 Introduction

The ever-increasing complexity of software systems together with the advent of new development methodologies, *e.g.*, extreme programming [8], tend to shift development costs from early stages, such as architecture and design, towards later stages, such as maintenance. In order to understand the software at these late stages, developers can benefit from additional information regarding its evolution, such as time and authors of code changes. This type of information facilitates team communication in collaborative projects, and also places investigations in the context of the entire project evolution. It allows, for example, to discover that problems in a specific part of the code appear after another part was changed. Such insight is easier to get when visualizing the context of the entire project evolution. In contrast, intensive debugging and runtime analysis is needed to get it from a single code snapshot.

In this chapter we present a novel approach to the visualization of evolution of source-code structure and attributes across the entire life span of a file. Typical questions that we try to provide answers to are:

- What code lines were added, removed, or altered and when?
- Who performed these modifications of the code?
- Which parts of the code are unstable?
- How are changes correlated?
- How are the development tasks distributed?
- What is the context in which a piece of code appeared?

The organization of this chapter is as follows. Section 5.2 details the structure of the visualized data. Section 5.3 presents the visual model we used to encode data. Additionally, visual image improvements and human interaction aspects are considered. To validate the proposed visualization methods and techniques we implemented them in a tool: CVSScan. This tool is aimed to support the program and process understanding during the maintenance phase of large software projects. A copy of the tool can be downloaded from [30]. Section 5.4 presents results of two case studies of using CVSScan to assess the evolution of files from real-life projects. Section 5.5 summarizes this chapter and outlines open issues and future directions of research.

5.2 Data Model

The history recordings of source code files can be retrieved from Software Configuration Management (SCM) systems. As presented in Section 3.4, the central element of a SCM system is a repository that stores all versions (*i.e.*, revisions) of a given file (see Section 3.4.1). Therefore, a repository R can be defined as a set of NF files:

$$R = \{F_i | i = 1, \dots, NF \in \mathbb{N}\}$$

and each file as a set of versions:

$$F_i = \{V_{j,i} | j = 1, \dots, NV_i \in \mathbb{N}\}$$

Each version of a file can be defined as a tuple containing its source code as an ordered set of lines, and a number of version specific attributes, for example the unique ID of the version, the author that contributed (committed) it to the repository, and the time when it was committed:

$$V_{j,i} = \langle \{\text{code lines}\}, ID, \text{author}, \text{date} \rangle$$

The visualization we propose considers files separately, so the file index is dropped in the following.

The SCM systems we used in our experiments (*i.e.*, CVS and Subversion) use a simple *entity similarity* function (Γ_{Line}) to record software evolution within one file (see

Section 3.4.1). This function is based on a tool similar to UNIX's `diff`, which reports the inserted and deleted lines of two consecutive file versions V_x and V_{x+1} . All lines not deleted or inserted in V_{x+1} are defined as constant (not modified), *i.e.*, $\Gamma_{Line} = 1$. Finally, lines reported to be both deleted and inserted in version V_{x+1} are defined as modified (edited), *i.e.*, $\Gamma_{Line} = k$, where $k \in (0, 1)$ is an arbitrary constant. Between any two arbitrary versions V_a and V_b , the similarity function is defined as a scalar composition of the similarity functions of all consecutive versions pairs between V_a and V_b (see Section 3.4.1).

Let l_i denote the i^{th} line of a version in some given context. An important concept for building our visualization of software evolution at line level is the *Global line set*:

Definition 5.2.1 *Global line set*

The global line set L associated with a specific line l_i in a file F is the complete set of lines l_j in all versions of F for which the entity similarity function between l_i and l_j is not null:

$$L(l_i) = \{l_j | \exists V \in F \text{ such that } l_j \in V \wedge \Gamma_{Line}(l_i, l_j) \neq 0\}$$

Next, we introduce three attributes that we use together with the SCM provided Γ function (Γ_{Line}), and attributes (version ID, author, date) for assessing the source code evolution at line level. The most important is the *Global line position*:

Global line position

$$G(L) : \{L | \exists l_i, V \in F \text{ such that } l_i \in V \wedge L = L(l_i)\} \rightarrow \mathbb{N}$$

with $G(L(l_i)) < G(L(l_j))$ if l_i comes before l_j in version V_x .

$G(L)$ induces a total order relation on the set of all global line sets associated with a file F , conform with the local order relations given by the line positions in each version of F .

$G(L)$ can be practically computed using a graph-based approach. For every global line set L , a graph node $N(L)$ is built. Nodes are created by scanning versions V_x in increasing order of x , and lines l_i in each version in increasing order of i . If lines l_i and l_{i+1} are consecutive in a given version, a directed arc is set from $N(L(l_i))$ to $N(L(l_{i+1}))$. Finally, when a node N is inserted between two other nodes N_A and N_B , an arc is set from any already existing node between N_A and N_B to N , to enforce a total order and not just a partial one. Figure 5.1 shows three versions of a file and their corresponding graph.

This graph is directed and acyclic, and gives a total order relation between all code lines of a file. The node corresponding to the global line set L before whose elements no other line existed during the whole project is the only one having only outgoing arcs. This *root* node (*e.g.*, node i in Figure 5.1) is labeled with zero and all other nodes are labeled with the maximal path length (defined as number of arcs) to the root node, by doing a topological sort of the graph [25]. Then, for every line l_i in every version holds that $G(L(l_i)) = \text{label}(N(L(l_i)))$. This gives a unique label to all code lines that belong to a file during development, keeps the partial line orders implied by the different versions in the project, and ensures that lines in different versions identified by `diff` as instances

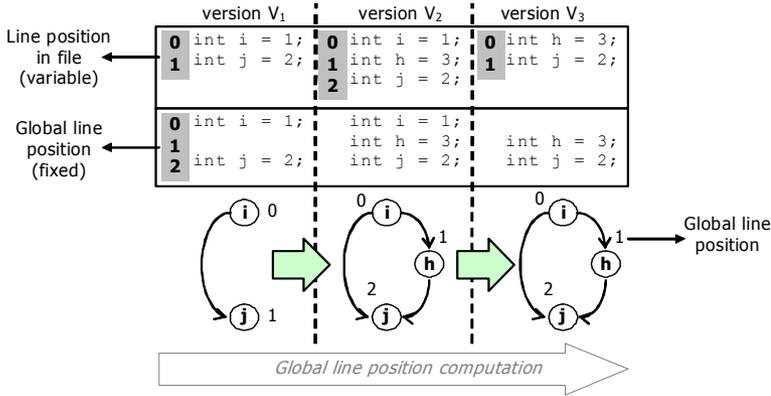


Figure 5.1: A graph-based approach for computing global line position

of the same global line have the same label. To keep the description simple, the syntagma *global position of a line* will next be used to refer to the global line position of the global line set associated with the given line. Next, the *Line status* attribute is introduced.

Line status

$$S(i, x) : \mathbb{N} \times \mathbb{N} \rightarrow \text{States}$$

with $S(i, x)$ = the state of the line at global position i in version V_x .

$S(i, x)$ is computed by comparing the line l_C at global position i in version V_x with lines in previous and following versions having the same global position i . Some values take into account also lines with global position $i + 1$ in previous, current and following versions. The status can be one of the following:

- *constant*:
 - if l_C exists, and
 - both lines at global position i in version V_{x-1} and V_{x+1} are identical with l_C when they exist
- *deleted*:
 - if l_C does not exist, and
 - there is a line l_P with global position i in a previous version V_y ($y < x$)
- *inserted*:
 - if l_C does not exist, and
 - there is a line l_N with global position i in a following version V_z ($z > x$)
- *modified by deletion*:
 - if a line at global position i in version V_{x-1} exists but differs from l_C , and
 - for the smallest j for which the line at global position $i + j$ in V_{x-1} exists it holds that the line at global position $i + j$ in V_x does not exist

- *modified by insertion*:
 - if a line at global position i in version V_{x-1} exists but differs from l_C , and
 - for the smallest j for which the line at global position $i + j$ in V_x exists it holds that the line at global position $i + j$ in V_{x-1} does not exist

- *modified*:
 - if the status cannot be classified as *modified by insertion* or *modified by deletion*, and
 - l_C exists, and
 - one of the lines at global position i in version V_{x-1} or V_{x+1} exists and differs from l_C

Figure 5.2 depicts the possible transitions of the values of the line status attribute for a given global line position.

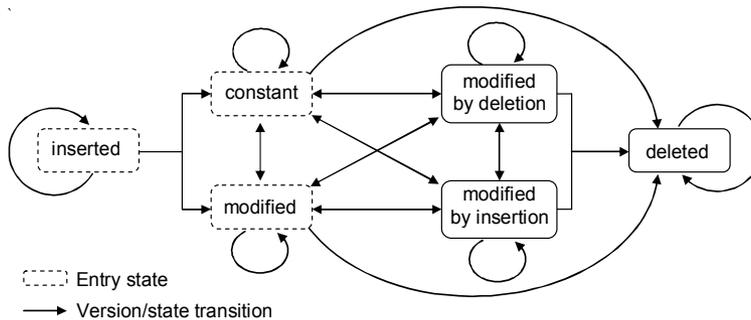


Figure 5.2: Line status transition diagram

Additional information can be extracted from the source code itself. When implementing our visualization we used a fuzzy parser with a customizable grammar to extract information such as blocks, comments and preprocessor macros. This type of information can be used to build a *Construct* line attribute:

Construct

$$C(i, x) : \mathbb{N} \times \mathbb{N} \rightarrow \text{Grammar}$$

with $C(i, x)$ = the syntactic construct to which that line at global position i in version V_x belongs to (e.g., comment, loop block, conditional execution branch), if such line exists.

Next we present the techniques we use to map the *Global line position*, *Line status*, *Construct*, and SCM provided attributes (i.e., version ID, author, date) to visual elements, in order to assess their evolution.

5.3 Visualization Model

The main focus of the software evolution visualization at line level is to enable the user to recover undocumented development knowledge about a file, including file structure, status and the associated developers network. Additionally, the user should be enabled to easily perform his assessments with a minimum cognitive overhead when investigating multiple representations of the data. To this end, we use a single-screen display for the visualization of the entire evolution of a file.

5.3.1 Layout and Mapping

Since software maintenance is mainly done at code level, we propose a line-based approach to visualize the software. Similarly to other line-based software visualization tools [37, 56, 47], we assume that developers are comfortable with visualizations that present the software in a spatial layout similar to the one they use to construct it. Consequently, we represent every line of code as a pixel line on the screen. We use a 2D representation, as advocated in Section 4.5. The main questions to answer in this case are how to layout the line representations in a plane, and how to use color for encoding attributes.

The layout we propose is different in two aspects from previous line-based layouts, such as the one proposed by Eick *et al.* [37]. Firstly, we use neither indentation nor line length to suggest code structure, but color in combination with a fixed-length pixel line for all code lines (Figure 5.3). This enables us to represent one version of a file on a very thin vertical stripe. Secondly, we visualize on the same screen all versions that a file has during its evolution, instead of all files in a project at a given time (Figure 5.4). The horizontal axis represents thus evolution in time and the vertical one the line position l_i . Each version is shown as a vertical stripe composed of horizontal pixel bars depicting lines of code (Figure 5.3b and c). Overall, this new approach trades revealing the length of code lines and their indentation off for offering a space-efficient filling to show files and code nesting level. This allows one to visualize more source code on the same screen. Our focus lays on one file at a time, in order to deliver a comprehensive view of its evolution, enabling users to make correlations between modifications in time.

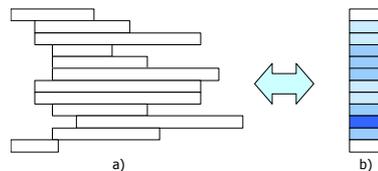


Figure 5.3: Line layout (a) SeeSoft; (b) our visualization.

Two main sampling strategies are possible on the time axis: version-uniform sampling and time-uniform sampling. In version-uniform sampling, each version is shown as a vertical stripe composed of horizontal pixel bars depicting lines (Figure 5.4b). This generates uniform incremental views on the evolution that are more compact and offer the same resolution both for *punctuated* evolution moments, *i.e.*, sharp variations of file

size denoting important changes [125], and for equilibrium periods. For time-uniform sampling, each line appears as a horizontal stripe, segmented according to its change moments (Figure 5.4c). This generates views that are more suitable for placing the file evolution in the development context of the project, with punctuated and equilibrium periods, but makes correlation more difficult in the punctuated evolution area, due to lack of resolution.

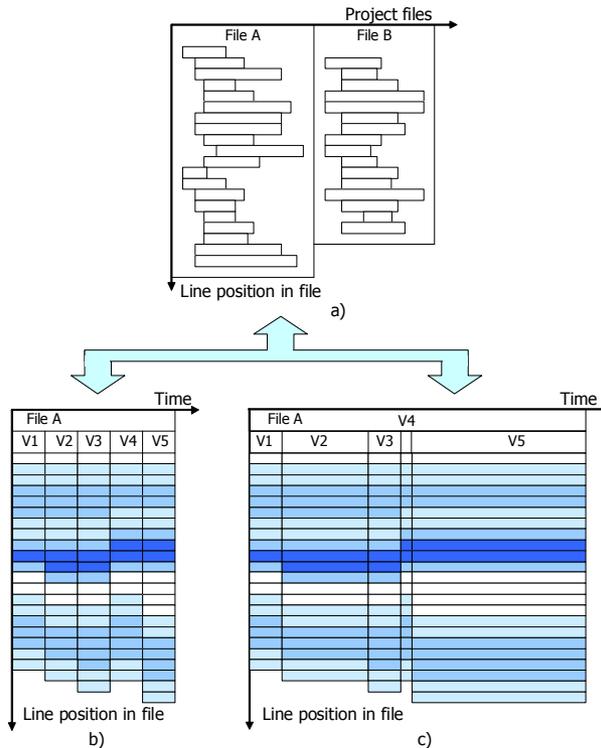


Figure 5.4: Use of horizontal axis in line-based visualizations: (a) files (SeeSoft); (b) time with version-uniform sampling (our visualization); (c) time with time-uniform sampling (our visualization).

For the vertical layout of lines within one version strip, we provide three alternatives. The first one, called file-based layout, uses as y coordinate the local line position l_i (Figure 5.5a). This layout offers an intuitive classical view on file organization and size evolution, similar to [37].

The second alternative, called line-based layout, uses as y coordinate the global line position $G(L(l_i))$ (Figure 5.5b). While this preserves the order of lines of the same version, it introduces empty spaces where lines have been previously deleted or will be inserted in a future version. In this layout, each global line has a fixed y position throughout the whole visualization. This allows easy identification of code blocks that stay constant in time, or get inserted or deleted. Consequently, it enables the identification of the continuation, insertion and deletion evolution patterns defined in Section 3.2.

However insightful, these two layouts do not offer both an intuitive view of a chosen version (*i.e.*, focus version) and a global overview of code deletion and insertion. Some investigation scenarios may require a view on the focus version with no empty spaces between lines, similar to the file-based layout, and at the same time a view that facilitates the identification of inserted or deleted code, similar to the line-based layout. To address these types of investigation scenarios we propose a third alternative: the interpolated layout, which interpolates between the previously presented two. We start from the bounding versions of the empty space interval with a line-based layout. Then we gradually decrease the y size of the empty spaces down to zero for the focus version (Figure 5.5c). In this way the focus version appears as a contiguous stripe containing no empty spaces, just as in the file-based layout, yet the information about inserted and deleted lines is still present in the image.

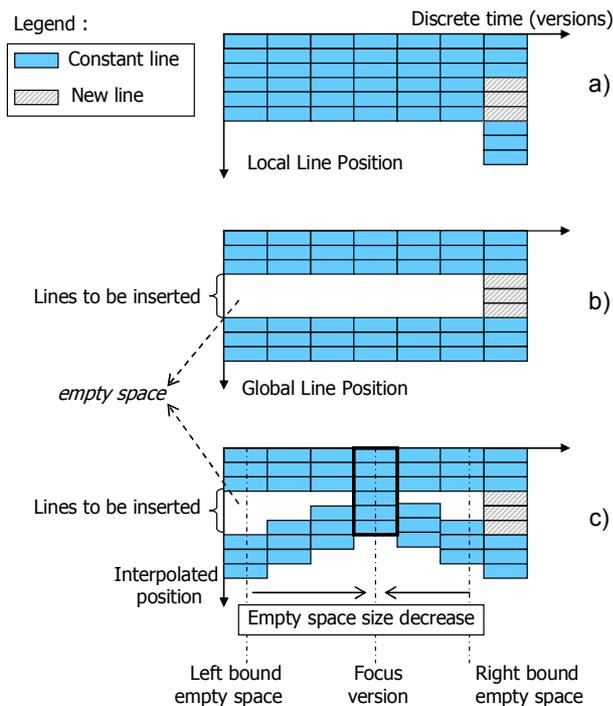


Figure 5.5: Vertical line layout: (a) file-based; (b) line-based; (c) interpolated.

In real-life software, a lot of code gets inserted and deleted during the project lifetime. The total y size of the focus version in the interpolated layout is considerably smaller than the sizes of the interval-bounding versions. The visual transition between their representations may thus become quite abrupt and difficult to follow. To make this transition smooth, we propose a number of solutions. First, we balance the representation by aligning the y midpoint of all versions with the image's y midpoint. The visual transition disruption caused by the vanishing empty spaces is now halved. Secondly, we use a configurable profile function for the size decrease of empty spaces, in order to distribute the visual transition disruption across the image's x axis. We use a weighted sum of exponential and hyperbolic tangent functions to compute the size of the empty spaces

(Figure 5.6a). Weight adjustment yields different visual disruption distributions. The profile function is applied on the x distance between the version containing the empty space and the focus version. Its result is normalized such that it equals zero when the empty spaces are in the focus version and the height of a pixel line when the empty spaces are in an interval-bounding version (Figure 5.6b).

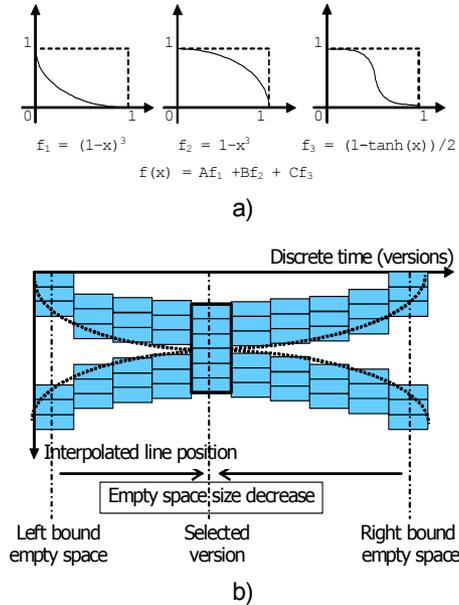


Figure 5.6: Profile function for empty space: (a) model; (b) example of balanced interpolated layout with asymptotical decrease of empty space size.

Next, we encode the *Author*, *Construct*, and *Line status* attributes defined in Section 5.2 by color (Figure 5.7). We use a customizable color map to indicate the status of lines in a given version (Figure 5.7a). We use a similar approach to encode constructs (*i.e.*, blocks, comments and references), and we modulate luminance to encode the block nesting level (Figure 5.7b). Finally, we use a fixed set of perceptually different colors to encode authors (Figure 5.7c). At each moment, one color scheme is active, such that the user can study the time evolution of its corresponding data attribute. When interesting patterns are spotted, one can switch to another scheme to get more detailed insight in the matter.

Figure 5.8 uses the approach that we propose to visualize the evolution of a file along 65 versions. Version-uniform sampling is used for the time axis. The three layout alternatives introduced above are illustrated. Color encodes line status: green denotes constant, yellow modified, red modified by deletion, and light blue modified by insertion respectively. Additionally, in the line-based and interpolated layouts (b and c), light grey shows inserted and deleted lines. The file-based layout (a) clearly shows the file size evolution and allows spotting the stabilization phase occurring in the last third of the project. Here, the file size has a small decrease corresponding to code cleanup, followed by a relatively stable evolution corresponding to testing and debugging. Yellow fragments correspond

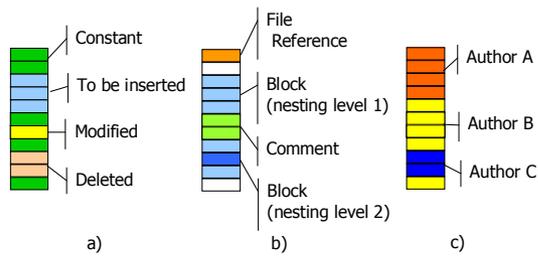


Figure 5.7: Attribute color encoding: (a) *Line status*; (b) *Construct*; (c) *Author*.

to areas that need reworking during the debugging phase. The interpolated layout (c) focuses on a particular version and indicates the points where code will be inserted in the following period.

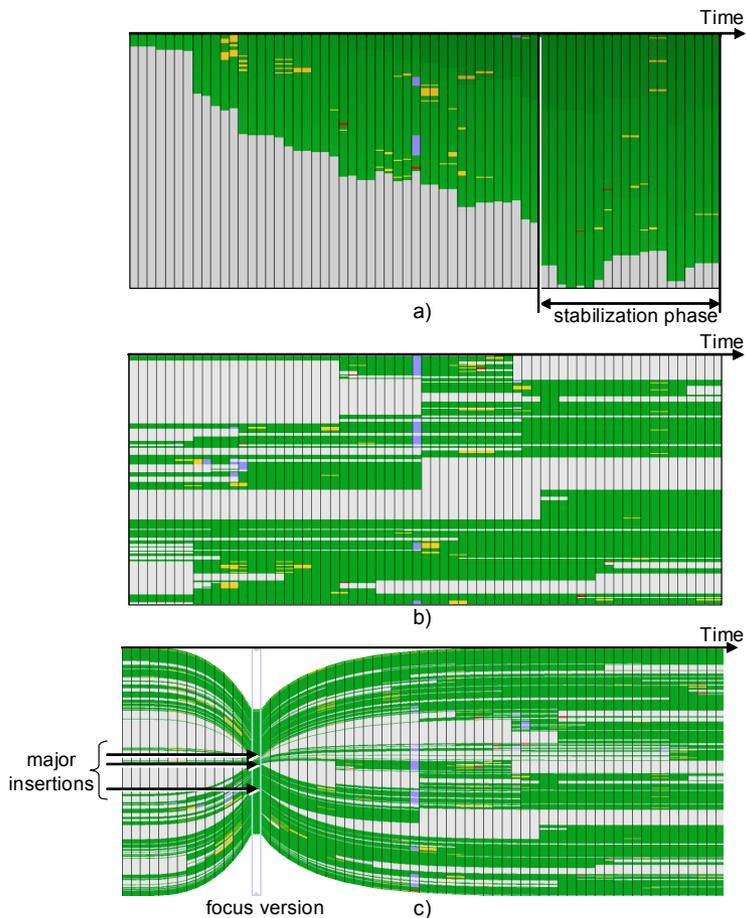


Figure 5.8: *Line status* visualization using a version-uniform sampling of the time axis and a: (a) file-based layout; (b) line-based layout; (c) interpolated layout.

Figure 5.9 illustrates different color encodings on a zoomed-in region of the line-based layout in Figure 5.8 (bottom). In Figure 5.9a, yellow is used to encode lines that are modified when passing from one version to another, as shown in the highlight. Yellow lines appear in pairs to make the identification of the change moment easier to detect, and to support mouse browsing during user interaction (see Section 5.3.4). Switching to the color scheme that encodes the *Construct* attribute (Figure 5.9b) enables the user to discover that the modified piece of code is in a comment, encoded by the dark green color. This means the modification does not actually alter the code functionality. Finally, the *Author* attribute (Figure 5.9c) shows the developer that performed the modification, here shown in purple.

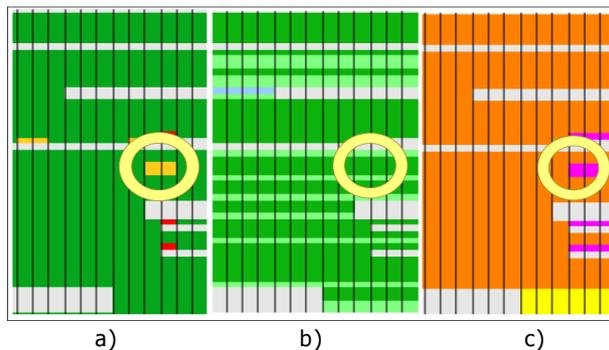


Figure 5.9: Attribute encoding: (a) line status; (b) construct; (c) author.

5.3.2 Multiple Views

A key factor in understanding the patterns revealed by evolution visualization is to correlate them with other information about the program. To this end, we offer two summarizing views (*i.e.*, metric views) in addition to the line-based visualization of code evolution presented so far, and also a novel text view on selected code fragments (Figure 5.10).

The metric views summarize data per version or per global-line and show the results with horizontal, respectively vertical color bars to complement the evolution visualization. Different data summarizing methods (*i.e.*, metrics) are available. For example, two horizontal metrics that we propose show, for each version, its number of lines and its author (Figure 5.11). A useful vertical metric shows the insertion moment of a code line for a given global line position, and gives a compact overview of the code development order in a file.

The code view offers a text look at the code. Users can select the code to be displayed by sweeping the mouse in the evolution view. Vertical brushing in the code evolution area scrolls through a version's code, whereas horizontal brushing over the line-based layout goes through a given line's evolution.

An important issue we addressed in the design of our visualization is how to correlate the code and evolution views, when the latter uses the line-based layout. The challenge was what to display when the user brushes over an empty space in the evolution view.

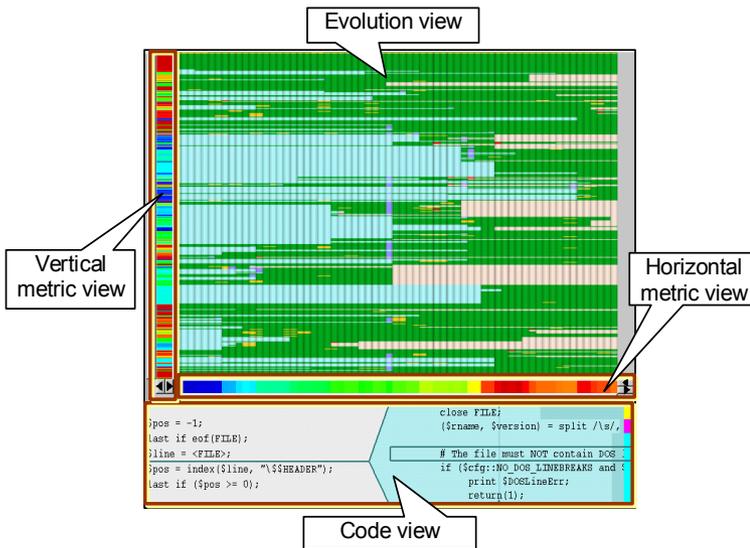


Figure 5.10: Multiple code views

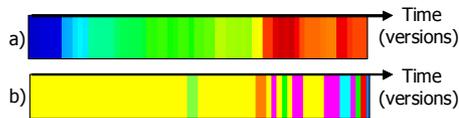


Figure 5.11: Metric views: (a) version size; (b) version author.

This space corresponds to *deleted* or *inserted* line status values (e.g., the light gray areas in Figure 5.8). Freezing the code display would create a sensation of scrolling disruption, as the mouse moves but the text doesn't change. Displaying code from a different version as the one specified by the mouse position, would have a negative impact on the context.

We addressed this challenge by a new type of code display. We used two text layers to display the code around the brushed global line position both from the version under the mouse and from versions in which this position does not refer to an empty space (Figure 5.12).

While the first layer (A) freezes when the user brushes over an empty region in the evolution view, the second layer (B) pops up, and scrolls through the code that has been deleted, or will be later inserted at the mouse location. This creates a smooth feeling of scrolling continuity during brushing. In the same time, it preserves the context of the selected version (layer A) and gives also a detailed, text level peek, at the code evolution (layer B). The three motions (mouse, layer A scroll, layer B scroll) are shown also by the highlights 1, 2, and 3 in Figure 5.16.

Another interesting challenge that we addressed was how to assess the code evolution shown by layer B. The problem is that lines of code located at consecutive global positions might not coexist in the same version. In other words, layer B consecutively displays code

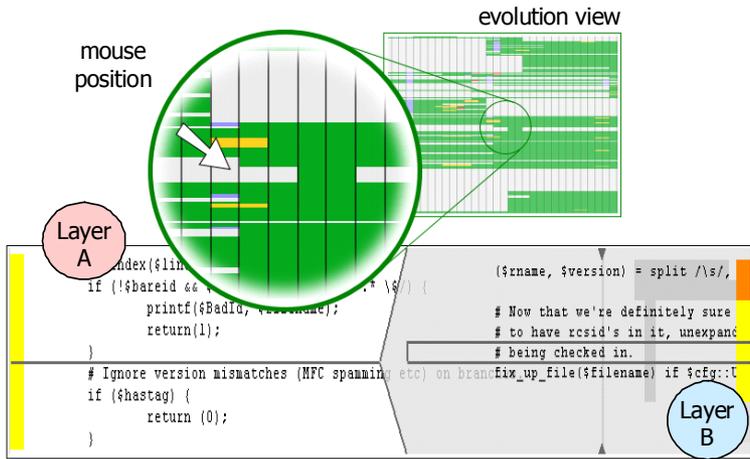


Figure 5.12: Two-layered code view

lines that may not belong to one single version. A way is needed to correlate this code with the evolution view. We achieved this by showing the lifetime of a line as a dark background area in layer B (Figure 5.13). Finally, we encoded the author of each line by colored bars near the vertical borders of the code view (Figure 5.12).

To summarize the technique presented above, the code view offers a detailed look on a specific global position in a selected file version, including information about its evolution and the developers that make it happen.

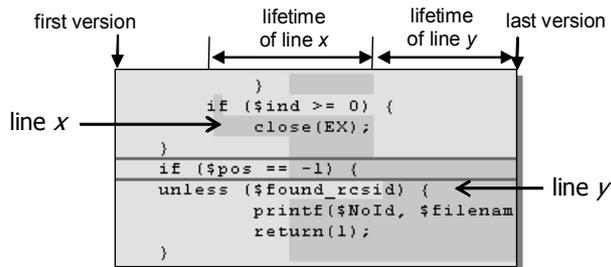


Figure 5.13: Code view, layer B. Line x is deleted before line y appears (*i.e.*, they do not coexist)

5.3.3 Visual Improvements

Real life software projects contain large files of thousands of lines. The resolution of commodity graphic displays is not sufficient to fit the entire file evolution on one screen, unless more lines share the same physical screen pixels. This raises the question how to represent code lines that share pixels such that the user gets a consistent, comprehensible and complete image of the file evolution.

We address this issue by a screen space antialiasing algorithm. We use antialiasing when the total number of lines to be displayed is larger than the available resolution. The algorithm computes the screen color of a number of overlapping lines by averaging their colors and weighting them according to their degree of overlap. That is, lines that fit inside one pixel location have a full weight, and lines that spread on more locations have a weight that equals the line percentage covered by the pixel location (Figure 5.14).



Figure 5.14: CVSScan antialiasing algorithm

Other alternatives to address this issue exist (see [64, 85]). One possibility would be to compute the line weight based on attribute values, such that the visibility of certain relevant lines can be guaranteed [85]. While this would help emphasizing lines based on their attributes, it would make structure based correlations more difficult across different display magnification levels, so more research is needed to find out whether and how well this alternative would work.

Figure 5.15 shows the benefits of this antialiasing scheme with a real-life example. We used an interpolated layout to depict the evolution of a 1350 line C code file along 100 versions. Color shows the line status attribute: dark blue = constant; light blue = inserted; and pink = deleted lines. Light blue and pink show thus empty spaces in the layout. The rightmost version is in focus in both cases. The lines in the beginning of the evolution appear to be interrupted when no antialiasing is used (Figure 5.15a). However, when the screen space antialiasing is enabled, lines become continuous and are easier to follow (Figure 5.15b).

5.3.4 User Interaction

In order to validate the visualization techniques proposed in this chapter we implemented them in a tool, CVSScan, which can be used to assess the evolution of source code files.

In addition to the visualization techniques previously described in this section, CVSScan offers a wide range of interaction means to facilitate the navigation of data. The repertoire of interactive exploration instruments provided by CVSScan is described below, using Shneiderman's perspective [96]. All instruments are designed to use a point-and-click approach, making the entire exploration possible only by the use of a mouse. A tool snapshot illustrating these mechanisms is shown in Figure 5.16.

As presented so far, the visualization we propose offers an intuitive **overview** on the evolution of a program file in a single 2D image, even for files whose number of lines exceeds the available screen resolution (Section 5.3.3). To get more detailed insight in a specific region of the evolution, CVSScan offers **zoom and pan** facilities. This enables the user to drill down to more detailed representations, in which the evolution of each line

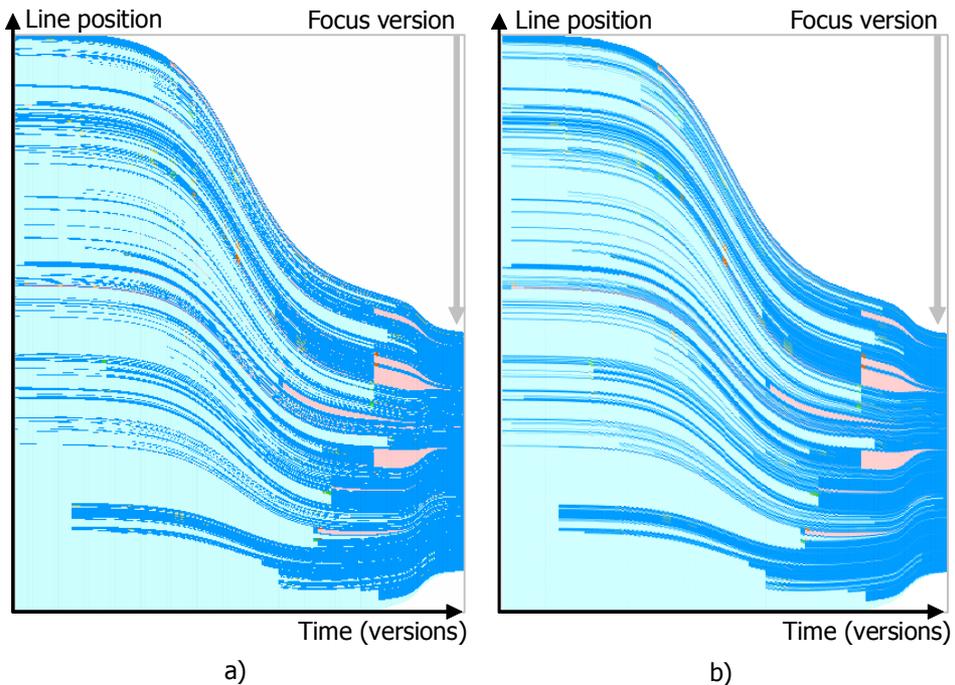


Figure 5.15: File evolution visualization using an interpolated layout: (a) without antialiasing; (b) with screen space antialiasing.

of code may be assessed. Two preset zoom levels are offered. These act as shortcuts to the global overview (fit all code to window size) and to the one-pixel-per-code-line level.

To support file evolution analysis from the perspective of one given version, CVSScan offers a **filtering** mechanism by means of which all lines that are not relevant are removed from the visualization (*i.e.*, lines that will be inserted after the selected version, or lines that have been deleted before the selected version). Filtering enables the user to assess a version, by clearly identifying the lines that are not useful and will be eventually deleted, and the lines that have been inserted into it since the beginning of the project. In other words, filtering provides a version-centric visualization of code evolution. Additionally, CVSScan gives the possibility to **extract** and select only a desired interval on which to study the file evolution. This mechanism is controlled by two sliders (shown in Figure 5.16, top) similar to the page margin selectors in word processors. By selecting an initial and a final version, one can hide the code that is not relevant (*i.e.*, code deleted before the initial version, or code inserted after the final one). This mechanism proved to be useful in projects with a long lifetime (*e.g.*, over 50 versions) in which one usually identifies distinct evolution phases that should be analyzed separately.

CVSScan enables the user to **correlate** information about the software evolution with specific details of the source code and overall statistic information. By means of metric views, users can visually get complementary information about lines (*e.g.*, the lifetime of a line at a given global position), or versions (*e.g.*, a version's author or size). The bi-

level code view (Section 5.3.2) offers **details-on-demand** about a code fragment: the text body, the line authors and the text evolution. The user can select the fragment of interest by simply brushing the file evolution area.

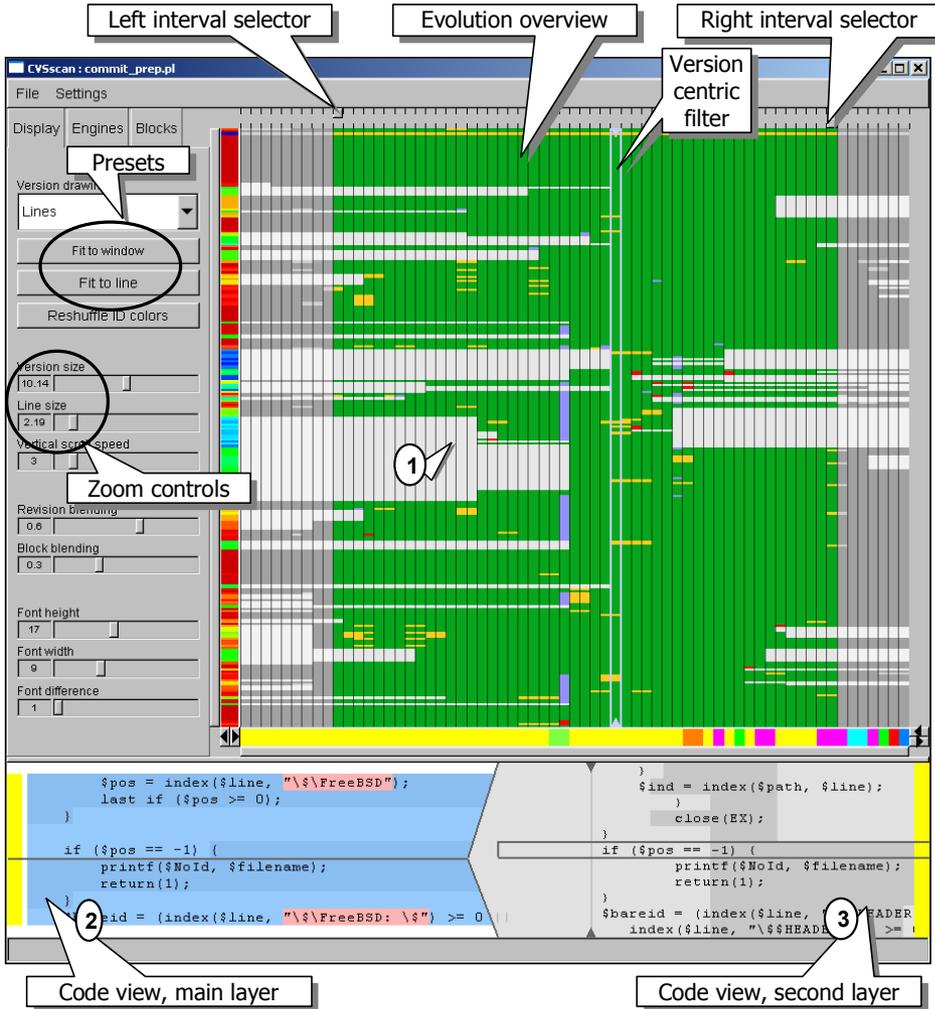


Figure 5.16: CVSscan tool overview. The file version and line number under the mouse (1) is shown in detail in the text views (2,3).

Although CVSscan is an exploration tool that does not alter the data it visualizes, it maintains a collection of state variables that could be externalized. This enables users to keep a **history** of their actions and lets them recover and reuse a specific visualization setting at a later time. In this direction, a simple extension that users suggested so far was to add an annotation facility by which developers can add their own comments, and visualize added comments, to a given version or line position.

The following section presents the results of a number of informal studies that show

how the visualization techniques and the interaction mechanisms presented in this section can be successfully used to investigate the evolution of files from real life software systems.

5.4 Use-Cases and Validation

The main target audience of the CVSScan tool is the maintenance community. They perform their tasks outside the primary development context of a project, and most of the times long after the initial development has ended. Therefore, the main activities a maintainer performs are related to context recovery, such as program understanding and team network building. CVSScan facilitates this process by visualizing file evolution from the perspective of different attributes and features, such as file structure, modifications, and authors.

In order to validate the visualization techniques and methods we implemented in CVSScan, we have organized a number of informal user studies based on the methodology proposed in [88]. The aim was to assess the visualization insight by analyzing the experiences of software maintainers when they investigate programs in whose development they did not participate, with no other support than CVSScan itself.

Below, we present the outcome of two such studies. In both cases, the users participated first in a 15 minutes training session. During the session, the tool's functionality was demonstrated on a particular example file. After that, each user was given a file for analysis, but no information about its contents whatsoever. A domain expert acted as a silent observer and assessed the correctness of the findings.

User study 1: analysis of a Perl script file

In the first case, we gave the user a script file from the FreeBSD distribution of Linux, containing 457 global line positions and spanning 65 versions. The user was familiar with scripting languages, but had no advanced knowledge about any of them. The user started CVSScan using the default file-based layout to visualize the evolution of file structure.

The user brushed first over the green areas in the evolution view: *These are comments, right? Let's see first what they say.*

He started to brush from the beginning of the file, choosing first the comments that spanned over the entire evolution. In the same time he read the code fragments displayed in the code view. *This is Perl. All Perl scripts have this path on the first line. This one looks like a file description. It reads that this script handles pre commits of files.*

Then, while brushing over the comment fragments (Figure 5.17a top/bottom): *These are annotated textual dividers: Configurable options, Constants, Error messages, Sub-routines, Main body. I use these too in my programs... Here are also some annotations.*

Further on, the user investigated also the large comment fragments that did not span over the whole evolution: *It looks like the implementation was either not completed or the developers left a lot of garbage. There are some code fragments over here that are*

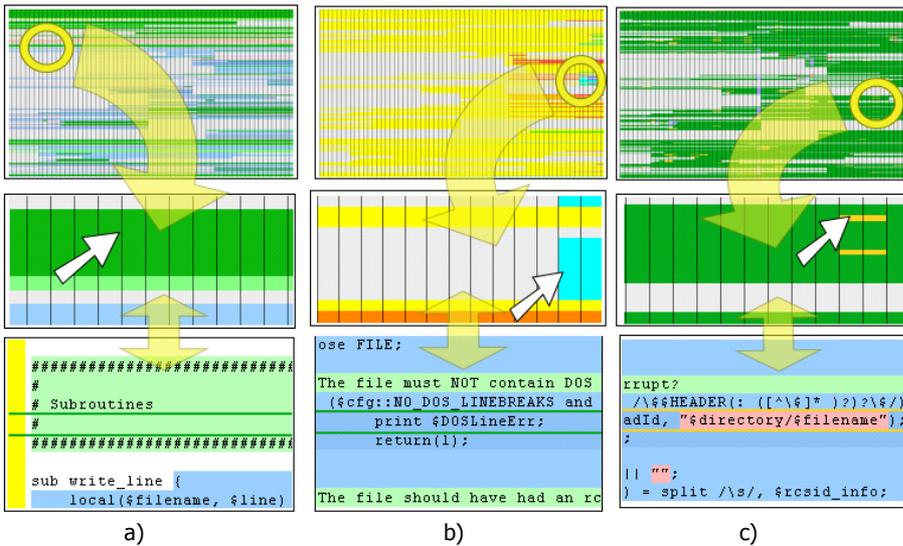


Figure 5.17: Case study 1 – Analysis of a Perl script

commented out.

The user next selected the last version and brushed over the *Subroutines* area: *It looks like these lines do not belong to any block. Here is a blank line before the write_line procedure. Here a blank line before exclude_file. So there are white lines before every procedure? Yes, indeed: check_revision, fix_up_file. So there are four procedures. It seems exclude_file is the most complex one as it has the highest nesting level.*

At this point, the user had a high-level understanding of the file structure. He started to make inquiries about the developers that had worked on the file. For that, he switched back and forth between the construct and author attributes using shortcut buttons: *The yellow developer, Dawes, did most of the work. However, the orange one, Robin, wrote that complex exclude_file procedure. He did that towards the end of the project, so probably that adds some extra functionality to the core. I see also that the cyan developer, Eich, did some significant work towards the end in the check_revision procedure (Figure 5.17b top bottom). It seems that his concern was to rule out files containing DOS line breaks... So this script doesn't handle DOS files?"*

The user then dismissed the authors that had only small contributions and switched to the line status visualization: *Apparently a major change took place in the middle of the project. It mainly affected the check_revision procedure.*

Then, selecting the version that followed the modified by insertion lines of the major change, the user started to concentrate on the areas where modifications took place: *I see a number of modifications between these two versions (Figure 5.17c top bottom). The first one replaces a file reference with a fully qualified name; the second does the same, the third too, the fourth, the fifth. Oh, they should have kept that file name in a separate*

variable! Here they tuned the regular expressions. Here they replaced a constant string with a variable.

The user continued to brush all areas where modifications appeared and tried to correlate them with the code and the authors that committed them. We interrupted the experiment after 15 minutes. At the end of the exercise, the domain expert considered the user was familiar with the overall organization of the file, the focus of each individual contributor, the places that had gone through important modifications and what these modifications referred to.

User study 2: analysis of a C code file

In the second case, we asked an experienced C developer to analyze a file containing the socket implementation of the *X Transport service layer* in the FreeBSD distribution of Linux. The file had 2900 global line positions and spanned across 60 versions. We provided this user with CVSScan. We enabled highlighting of C grammar and preprocessor constructs, such as `#define` and `#ifndef` (see Section 5.2).

The second user started the tool in the default mode too, and tried first to look for commented fragments: *This is the copyright header, pretty standard. It says this is the implementation of the X Transport protocol, pretty heavy stuff... It seems they explain in this comments the implementation procedure.*

The user next switched his attention to the compiler directives: *A lot of compiler directives. Quite complex code, this is supposed to be portable on a lot of platforms. Oh, even Windows.*

Next, the user started to evaluate the inserted and deleted blocks: *This file was clearly not written from scratch, most of its contents has been in there since the first version. Must be some legacy code... I see major additions done in the beginning of the project that have been removed soon after that... They tried to alter some function calls for Posix thread safe functions (Figure 5.18a top bottom)... I see major additions also towards the end of the project... A high nesting level, could be something complex... It looks like code required to support IPv6. I wonder who did that?*

The user switched then to the author visualization: *It seems the purple user, Tsi, did that (Figure 5.18b top bottom). But a large part of his code was replaced in the final version by... Daniel. This guy committed a lot in the final version... And everything seems to be required to support Ipv6. The green user, Eich, had some contribution too... well, he mainly prints error messages.*

Eventually, the user switched to the evolution of line status and used the predefined *Fit to line* setting to zoom in. *Indeed, most work was done at the end... Still, I see some major changes in the beginning throughout the file... Ah, they changed the memory manager. They stepped to one specific to the X environment I assume. All memory management calls are now preceded by "x" (Figure 5.18c top bottom)... And here they seem to have given up the TRANS macro.*

The user spent the rest of the exercise assessing the modifications and the authors that committed them. We interrupted the experiment after 15 minutes. At the end, the

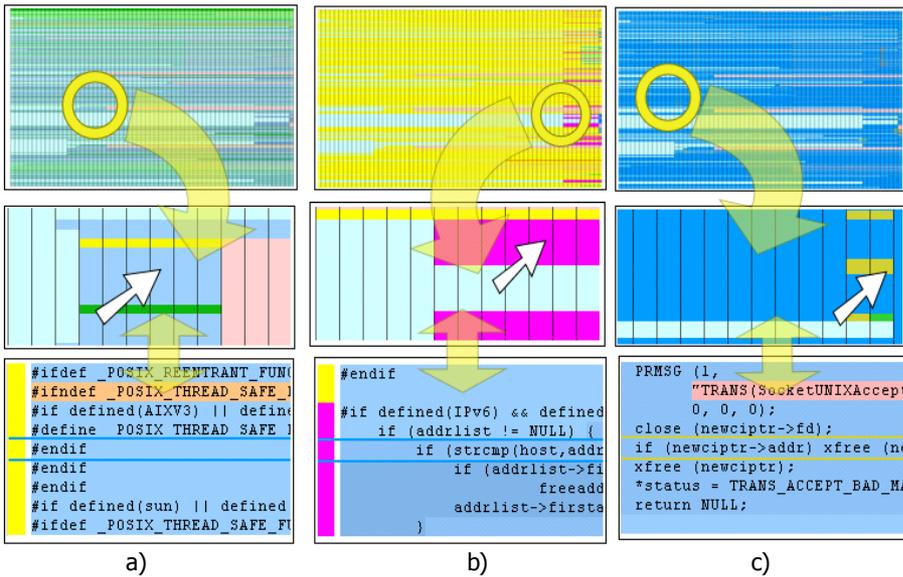


Figure 5.18: Case study 2 – Analysis of a C code file

domain expert considered the user did not have a very clear image of the file's evolution. However, the user managed to discover that the file represented a piece of legacy code adapted by mainly two users to support the IPv6 network protocol. He also pointed out a major modification: the change of the memory manager.

5.5 Conclusions

In this chapter we have presented a new approach for visualizing software evolution at line level, using line-oriented displays. This novel approach offers multiple correlated views on the evolution of a source code file. Dense pixel displays are used to show the overall evolution of code structure, semantics and attributes, and they are integrated in an orchestrated environment to offer details-on-demand. Additionally, we have introduced a novel type of code text display that gives a detailed and intuitive view on both the composition of a fragment of code and its evolution in time.

To validate the presented visualization methods and techniques, we implemented them in a tool for source code evolution assessment: CVSScan. We used this tool to perform a number of user studies on data extracted from real-life SCM repositories. In this chapter we have presented the outcome of two of these experiments. Although informal, the studies show that the line-based evolution visualization of code supports a quick assessment of the important activities and artifacts produced during development, even for users that had not taken part in any way in developing the examined code. To reduce the subjectiveness of these findings, more experiments have to be organized using a larger number of subjects and assessors on the same software input.

In both studies the domain experts liked the tool and considered it could be used to gain even more insight in the system under investigation. They expressed their interest in assessing the tool usability for discovering more detailed information. In this respect, they considered that the relatively short examination time (*i.e.*, only 15 minutes) did not allow users to consolidate their knowledge about the system and make more advanced correlations. Consequently, future experiments that would assess these issues should increase the examination time and should use subjects that are familiar with the investigated software.

The user study subjects valued mostly the compact overview coupled with easy access to source code. These enabled them to easily spot issues at a high level and then get detailed source code information. To the best of our knowledge, this is the first approach that allows users to navigate code evolution at line level using an overview of it. Additionally, the approach we propose is relatively generic as it may be applied to study the evolution of any line-based structure.

A weak point of our visualization so far is the accuracy of the `diff` operator used to discover differences between versions of the same file. We used the `diff` operator provided by CVS and Subversion repositories. Therefore, the accuracy of our visualization depends on the relatively simple heuristics behind this operator, which can lead to data misinterpretations. A significant improvement would be to use a `diff` tool with support for semantic comparisons.

Another open issue of our visualization is the evaluation of the interpolated layout. During the studies that we organized, the subjects did not use this type of layout in their assessments. A possible explanation for this would be the short duration of the experiments and the absence of task specifications that would limit the search horizon of the user. This distracted the subjects from performing a version-centric analysis of evolution, which would benefit from the support of a visualization using the interpolated layout.

Chapter 6

Visualizing Software Evolution at File Level

In this chapter we investigate how software developers can get insight in software evolution of entire projects. To this end, we propose a number of new techniques for visual mining of project evolution. Central to our approach is a file-based evolution visualization, where each project is shown as a set of horizontal stripes depicting files along the time axis. We propose two mechanisms for interactively reordering and organizing the stripes in this display. We also propose a new multivariate visualization technique that enables complex correlations. To this end, we use a combination of color and textures to depict up to three artifact attributes at the same time in one view using the same spatial layout. We use interaction to extend the correlation capabilities to four or more attributes. To reduce the complexity of evolution visualizations we use clustering, and we compare two methods to generate relevant abstraction levels in a hierarchical clustering data set. Additionally, we introduce a novel widget, i.e., the cluster map, which visualizes all partitions in a hierarchical clustering set and supports users when choosing a partition to be visualized. We demonstrate the efficiency of the proposed methods and techniques with a number of analysis experiments that we performed on existing real-life systems.

6.1 Introduction

Software evolution assessments provide useful information about the development context of a project. Effective use of this information can greatly help maintainers understand and manage evolving projects. Additionally, project specific evolution patterns may be identified during assessment. These could support planning of project activities and could help improving the software quality.

The evolution visualization technique at line level proposed in Chapter 5 offers insight in the evolution of one file at a time. However, for evolution assessment, insight from correlations across the boundary of one file is required. In this chapter we present a set of

new techniques for visually assessing the entire evolution of software projects using the evolution information contained in SCM systems. Typical questions we target with these techniques are:

- What is the project-wide activity, *i.e.*, when have files been created, modified, and by who, and how did this activity evolve during the project?
- Which are the project areas of high(est) activity?
- How are development tasks distributed among the programmers?
- Which are the project files that belong and/or are modified together?
- How well do the conceptual and functional organization of a project match the actual folder structure?

The structure of this chapter is as follows. In Section 6.2 we present a model of the evolution data that we visualize. Section 6.3 details the visual layout mechanisms we used for evolution visualizations and for correlations with other results of project evolution analysis. Additionally, several interaction techniques are proposed to support the visual mining of the evolution data. To validate the techniques that we propose, we implemented them in a tool for software evolution assessment of entire projects: CVSgrab. This tool seamlessly combines SCM data extraction with analysis and visualization. A copy of the tool can be downloaded from [29]. Section 6.4 presents several use-cases that illustrate the use of CVSgrab for investigating the evolution of industry-size projects. Section 6.5 summarizes this chapter and outlines open issues for future research.

6.2 Data Model

The software evolution visualization at file level that we propose builds on the data model outlined in Section 3.4. A hierarchical organization of software in folders and files is assumed, together with a similarity function Γ_{file} at the level of files. Consequently, the visualized entities are file revisions. Additionally, we use the same Γ_{file} function as the one presented in equation 3.4.1.

In the proposed visualization, the evolution of files is correlated with the evolution of a number of file attributes. These are either directly available from SCM repositories (*e.g.*, file creation time, author ID, file type, file size), or are computed based on the stored evolution recordings (*e.g.*, presence of a specific word in the associated revision log). Next, we propose a visualization model that enables evolution correlations across all files of a project, and with multiple file attributes at the same time.

6.3 Visualization Model

The purpose of software evolution visualization at file level is to enable users to obtain insight in the project-wide structure and to see correlations between attributes across the

evolution of all files implementing a system. To this end, the visualization that we provide offers a single-screen display of the entire evolution of a given project. Similarly to the work presented by Eick *et al.* in [37] and in Chapter 5, we use a 2D code-centric approach to visualize the software evolution. As a new element, we interactively present the entire evolution of complete projects. This enables users to actively use visualization for mining the history of software systems.

6.3.1 Layout and Mapping

One of the biggest challenges of visualizing the evolution of complete software projects is scale. The approaches introduced by Lanza in [69] and by Wu *et al.* in [125] are the only ones we are aware of that scale well for visualizing the evolution of industry-size projects. Both techniques use a fixed vertical ordering of the entities (classes and files respectively). This fixed ordering, however, does not specifically help to find evolution based correlations.

We propose a novel approach for visualizing complete projects with a flexible entity layout that can be interactively modified by users to suit specific analysis scenarios. Similarly to the approach of Wu *et al.* [125] we visualize projects at file granularity level. Every file is drawn as a fixed height horizontal stripe made of several segments (Figure 6.1). Each segment corresponds to one version of that file. Segments are ordered according to creation time and their length is scaled with the lifetime of the respective version. Compared with the approach we propose in Chapter 5, this corresponds to a time-uniform sampling of the horizontal axis. In this case, however, we do not offer a version-uniform sampling alternative. The main obstacle in this direction is the technical complexity of building a scalable implementation of such sampling for entire projects. Nevertheless, a version-uniform sampling could offer better resolutions when assessing the moments of *punctuated* evolution (see [125]) so more research is needed in this direction.

Version segments can be colored to show various file version attributes. First, we can show the author that committed the respective version by mapping the author ID to a unique hue (Figure 6.1 top). This helps evolution correlations based on both activity and the authors' network. Alternatively, color can show the state of the version in the context of a complete project, *i.e.*, file not created yet, before last version, last version (Figure 6.1 bottom). This provides a simpler image that focuses specifically on activity events. Other alternatives for color encoding are file attributes directly available from SCM repositories (*e.g.* file type, file size) or computed based on the SCM recorded information (*e.g.*, presence of a specific word in the associated revision log). For all alternatives, we use geometric shaded cushions [111] to emphasize the version segments and to separate vertically stacked file stripes. Also, we draw the commit moments themselves as thin vertical lines between the version cushions.

We build complete project visualizations of software evolution by stacking individual file stripes on the vertical axis so they share the same time scale and use the same color encoding. In contrast with the approaches of Lanza [69] and Wu *et al.* [125], we do not fix the vertical axis ordering, but allow (and encourage) users to interactively change the layout to target specific analysis needs. We describe next two mechanisms to achieve this goal: sorting and clustering.

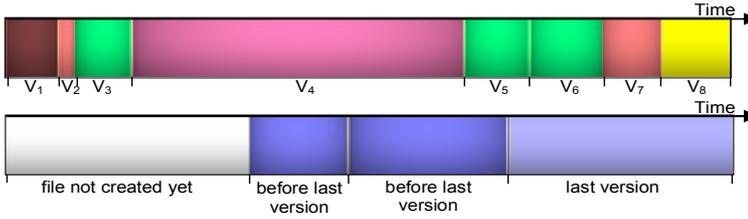


Figure 6.1: File evolution representation. Color encodes user identity (top) and activity (bottom).

Sorting

Sorting allows identifying how a relevant project metric is distributed across a set of files. Files are ordered along the vertical axis according to that metric's values. Similarly to the TableLens system [92], we propose several metrics that generate alternative layouts of the project evolution: creation time (similar to [125]), alphabetic order (similar to [69]), activity measure (*i.e.*, number of versions), and evolutionary coupling measure (see [129]). Other metrics can also be of interest, depending on the specific evolution assessment scenario.

The term *evolutionary coupling* in the context of software evolution was first introduced by Zimmermann *et al.* [129] and it is used to describe the simultaneous change of software entities during development. The evolutionary coupling metric that we propose is a function $S : \{F_i\} \times \{F_i\} \rightarrow (0, 1]$, where $\{F_i\}$ is the set of all files in the system. Given some file of interest F_1 , we measure the similarity $S(F_1, F_2)$ between its evolution and that of another file F_2 as a number in the interval $(0, 1]$. The higher the number, the higher the evolution similarity, and consequently, their evolutionary coupling. To define $S(F_1, F_2)$, we introduce first the notions of commit neighborhood N_K and evolution correspondent τ . Let V_i be the set of commit moments for all versions of a file F_i . Then $N_K : V_1 \rightarrow V_2^*$ is a mapping that assigns to each element t of V_1 a set of elements $V_2^* \subseteq V_2$ that are in a time vicinity of K time-units from t :

$$N_K(t) = \{u | u \in V_2, |u - t| < K\}$$

Next, $\tau : V_1 \rightarrow V_2 \cup \{\infty\}$ is a mapping that assigns to each element t of V_1 the closest element u from $N_K(t)$, if such an element exists, or ∞ (infinity) otherwise:

$$\tau(t) = \begin{cases} u_{min} & |N_K(t) \neq \emptyset \wedge |t - u_{min}| = \min |t - u|_{u \in N_K(t)} \\ \infty & |N_K(t) = \emptyset \end{cases}$$

We define now the evolutionary coupling $S(F_1, F_2)$ of files F_1 and F_2 as the symmetrized sum of inverses of the time difference between all commit moments in a file and their evolution correspondents in the other file:

$$S(F_1, F_2) = \frac{1}{2|V_1|} \sum_{t \in V_1} \frac{1}{\sqrt{|t - \tau_1(t)| + 1}} + \frac{1}{2|V_2|} \sum_{u \in V_2} \frac{1}{\sqrt{|u - \tau_2(u)| + 1}}$$

where τ_1 is the evolution correspondent from V_1 to V_2 , and τ_2 is the evolution correspondent from V_2 to V_1 .

This measure says that files that are changed at similar moments, are more similar than others from an evolution perspective. The underlying idea, which can be checked as correct in many large software projects, is that files which depend on each other, either via explicit data, call structures or otherwise, must (and will) be changed together to maintain the desired system invariants. Thus, evolutionary coupling is related to interface or implementation interdependencies. Compared to the approach proposed by Zimmermann in [129] we look for similar commit moments, but we do not try to group these in transactions. This enables us to correlate files that are developed by different authors and have different comments attached, but are nevertheless highly coupled.

Using $S(F_{ref}, F_i)$ permits us to sort all files F_i of a project according to the temporal similarity in change behavior they have with respect to a given reference file F_{ref} . Figure 6.2 shows an example of the proposed evolutionary coupling measure used to sort files on the vertical axis. The evolution of 23 files is colored by activity, as described for Figure 6.1. Yellow lines show commit moments. The topmost file is the reference file F_{ref} chosen by the user, the other files are vertically sorted on decreasing evolutionary coupling with respect to F_{ref} . This image allows us to easily find files that have a similar evolution as the reference file.

Clustering

The second mechanism we propose to layout files is clustering. Industry-size projects can contain thousands of files. Following the evolution of each individual file and correlating it with the evolution of the others is simply too complex. Clustering lets users group files that are similar from a certain perspective. Clustering has two roles. First, it lets users look at less data to investigate evolution correlations, reducing the complexity problem. Second, it offers system decomposition, facilitating the software understanding process when no such decomposition is available.

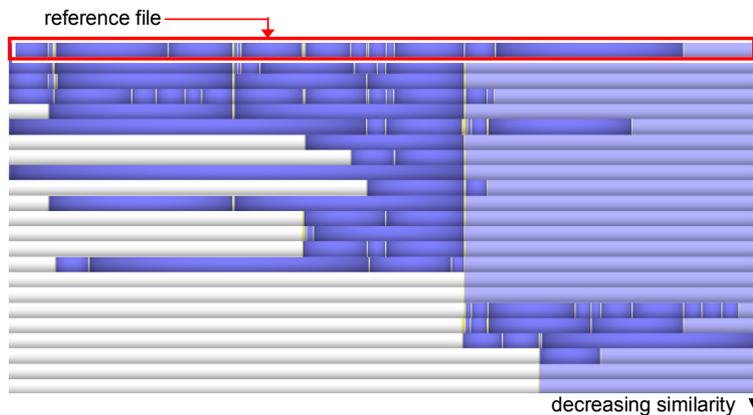


Figure 6.2: Sorted files layout based on an evolutionary coupling measure

Two issues must be addressed when implementing a clustering mechanism. First, we must provide a meaningful similarity measure. Second, we must provide a method for grouping similar files. We use as similarity measure the evolutionary coupling metric previously introduced for the sort mechanism, and a bottom-up agglomerative clustering based on average-link to group similar files as explained in [41]. We start with the individual files and recursively group the two most similar ones in a cluster, until a single cluster is obtained, creating thus a cluster tree.

When a new cluster is constructed, a set of imaginary commit events is derived for it based on the commit events of its two children. This set is further used in the clustering process, when computing higher level clusters. Let V_1 and V_2 be the sets of all commit events of the two children of a cluster. Then we compute the set $V_{cluster}$ of all commit events associated with the cluster using the formula:

$$V_{cluster} = \bigcup_{t \in V_1} \left(\frac{t + \tau_1(t)}{2} \right) \cup \bigcup_{u \in V_2} \left(\frac{u + \tau_2(u)}{2} \right) \setminus \{\infty\}$$

where τ_1 is the evolution correspondent from V_1 to V_2 , and τ_2 is the evolution correspondent from V_2 to V_1 . Otherwise stated, we populate the cluster with the averages between all commit events that have a commit neighborhood and their evolution correspondents. This enables the propagation of similar commit events to higher level clusters.

After the cluster tree is constructed, the user can choose to view the project at a desired level of detail by using a visualization that visits the tree and draws the clusters that match a certain criteria. For instance, one could visualize all roots of the intermediate cluster trees that were present during the k^{th} step of clustering algorithm, giving a decomposition of the system in $N - k$ clusters, where N is the total number of files in the system. Although our clustering may be computationally more intensive than other techniques, *e.g.*, k-means [41], it provides a simple, automatic and deterministic way to identify similar entities. We visualize the clustering results using colored and shaded cushions. Clusters are rendered as semitransparent rectangles atop of the file stripes, textured with plateau cushions [72], *i.e.*, luminance signals that increase parabolically close to the margins and have a constant (plateau) value in the middle. We use alternating hues, for instance, blue and red, for neighbor cushions. Due to the semi transparency of the cushions, these hues blend with the file stripes. The alternating hues effectively help visual segregation of clusters depicted by cushions. For example, Figure 6.3 shows cluster cushions with and without alternating hues.

However, alternating hues alone may not be sufficient for visual segregation. When a rich color encoding is used for the file stripes, for instance, the author-id color encoding, we must minimize its interference with the cushion hues. A too soft cushion hue blending over richly colored file stripes yields a poor visual separation of clusters in the border regions.

Figure 6.4 presents an example. It depicts the evolution of 10 files with color-encoded author-id. Three clusters are also shown, the first one containing the first four files, the second containing the following two, and the last containing the remaining four. Figure 6.4 top uses a color-only blending scheme to segregate between clusters. However, the visual transitions between clusters are not obvious. One could easily interpret the color change

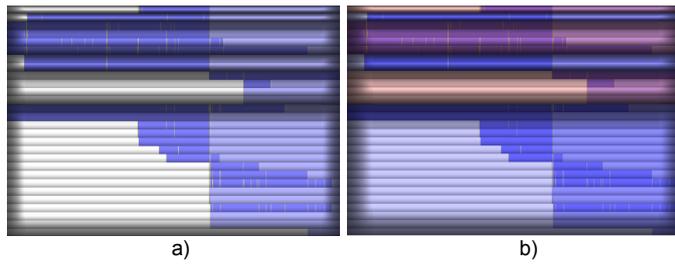


Figure 6.3: Cluster segregation using plateau cushions: (a) without alternating hues; (b) with alternating hues.

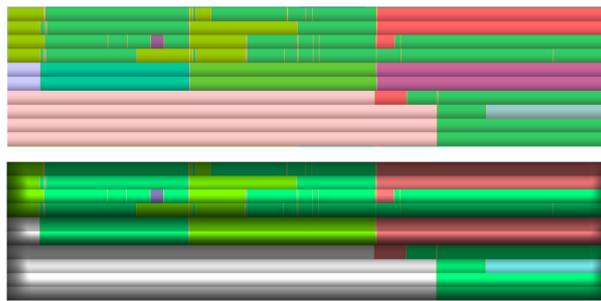


Figure 6.4: Cluster segregation: (top) color blending only; (bottom) with plateau cushions.

as author-id change and not as another cluster. In contrast, Figure 6.4 bottom uses plateau cushions and one can now easily identify the three clusters. We experimented with different cushion profiles, such as purely parabolic [111]. However, the design presented above was the most visually pleasing and effective of the studied ones.

By combining sort and cluster operations, we can interactively build visualizations of project evolution that suit specific analysis needs. Figure 6.5 shows the evolution of 28 files from a real project, FreeDesktop [46], using an interactively built layout. The alternating blue-red hue blending and plateau cushions introduced above are used to segregate clusters. Files are colored on activity: white (pink or light blue after hue blending) means file not created yet, dark blue (dark blue or magenta after hue blending) means before last version, light blue (light blue or magenta after hue blending) means last version. Yellow lines show commit moments. Six clusters emerge, each containing files with a similar evolution. Within each cluster, files are sorted according to their creation time. This image immediately shows files with similar behavior. The strongly related files in cluster 1 are: `Glyph.c`, `Picture.c`, `Xrender.c`, `Xrender.h`. At detailed inspection, we discovered that these files contain code of the project's image generation engine. This confirms the correlation between evolutionary coupling and conceptual similarity. A second finding is that files with a strongly coupled evolution, *i.e.*, clusters 1 and 2, have also a similar creation time and this time is close to the project beginning. Files that are created later seem to be less connected (cluster 3). This may be an indication that the system's core functionality, developed in the beginning of the project, is found in clusters 1 and

2. Concluding, the interactively built layout in Figure 6.5 enables user-driven cross-file correlations based on similar evolution and the creation time metric. Such correlations do not address only the development process assessment. As illustrated by this example, they may also bring insight in the structure and organization of the project, a key requirement in the maintenance phase of many projects. The interactive layout technique we propose enables the user to combine clustering with a refined sort operation, *i.e.*, equal values in one sort criterion may be further ordered using another metric, to adapt the visual mining process to specific needs. Useful correlations can be obtained by comparing the results of different sort operations.

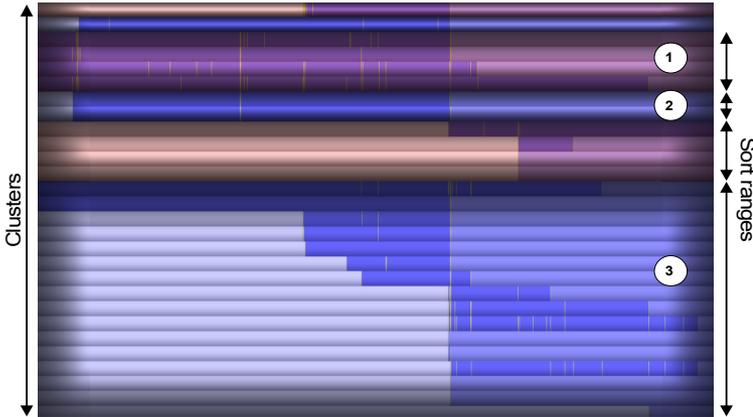


Figure 6.5: Interactively built layout using sort and clustering operations

6.3.2 Metric Views

To further extend the correlation capabilities of our interactive layout in this direction, we use metric views, *i.e.*, narrow information bars along the main evolution visualization area that summarize file and evolution information. These views use simple encoding techniques, *e.g.*, 1D graphs and color maps, to show one-dimensional metric data in a very small space. To enable finding correlations, metric views share their main axis with one of the axes of the main visualization. Vertical views visualize per-file computed quantities (*i.e.*, evolution metrics), and horizontal views visualize time-varying project metrics (*i.e.*, version metrics). In the vertical metric view we show the various metrics used for sorting, *e.g.*, the file creation time, alphabetical order, activity measure, and evolutionary coupling with respect to a reference file. In the horizontal metric view, we visualize the project-wide activity measure, *i.e.*, total number of files updated in a given period. Other metrics can be presented in these views, depending on the specific analysis scenario.

Figure 6.6 shows the evolution of 68 files from a large project (the VTK library [118]) using the same color encoding as in Figure 6.2, *i.e.*, activity based, and sorted on creation time. The vertical metric view shows the file activity as a 1D bar graph. The horizontal metric view shows the project wide activity. By correlating the main layout with the vertical metric view, we see that file creation time does not fully determine the file activity. Two activity hotspots are identified. They correspond to groups of files that appeared later

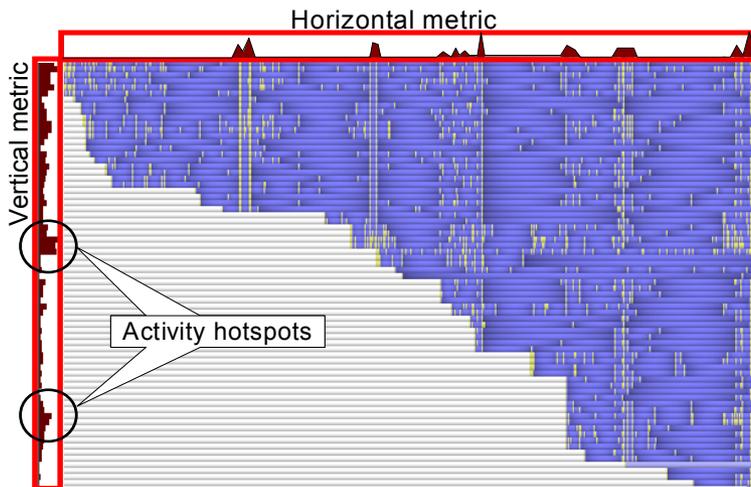


Figure 6.6: Metric views: vertical encodes file activity; horizontal encodes project-wide activity.

in the project but had high activity, so they might contain important and/or problematic functionality. We validated this hypothesis against the knowledge of an expert VTK user, and it proved to be consistent with reality. Concluding, the correlation of the interactively built layout with the metric views enables the user to easily construct hypotheses about qualitative aspects of a project based on its evolution. While this does not immediately guarantee a valid system assessment, it represents a solid starting point for further investigation and facilitates the understanding process during the maintenance phase of software projects.

6.3.3 Multivariate Visualization

Software evolution data is multivariate. Every version of a file has a number of assigned attributes that characterize it, such as version ID, commit time, author ID, and author commit log message, which are explicitly stored by SCM systems (see Section 3.4). Atop of these explicit attributes and of the source code itself, many new attributes can be defined, for instance the code source size, the presence of a given word in the author comment, the membership to a given software release, and so on. To assess the evolution of a software project, the distribution of such attribute values can be visualized, for instance using the basic visualization model described in Section 6.3.1. Each such visual distribution gives a viewpoint on the project evolution. More insightful information in the evolution can be obtained through correlations across multiple view points. However, this typically requires visualizing the distribution of more attributes simultaneously. Moreover, the correlation making process is a dynamic activity, so a way to define, customize, and select usage scenarios is needed. To achieve these goals, we had to address several challenges. First, the visualization model we propose maps real-life projects of thousands of files with hundreds of versions to small-sized pixel strips. We had to find effective ways to

map several attributes on this small space. Secondly, we had to find ways to enable users to construct the attribute mapping functions intuitively and quickly.

Texture Synthesis for Attribute Visualization

To address the first challenge, we chose to depict multiple attributes on the same (small) space using a combination of color and hand-designed textures. Our approach resembles the one proposed in [60]. However, there are important differences. Holten *et al.* use a parameterized synthetic texture to encode one attribute besides the one encoded by color. Their texture model allows easy building of tiling textures that do not perceptually interfere with other shapes depicted in the visualization, hence do not artificially grab attention. However, this approach seems to be less suitable to encode more attributes at the same time and over the same quite small screen space. The inherent irregular texture aspect, due to the noise-based synthesis method, makes it difficult to distinguish between two (or more) superimposed patterns, used to map two (or more) attributes. We propose a different approach that allows encoding two or three attributes via superimposed, yet visually distinct, textures. For this, we give up the generality of the irregular texture synthesis proposed by Holten *et al.* We choose several hand-designed texture patterns, and encode attribute values in the pattern magnification factor. A careful design and selection of patterns by hand ensures that these interfere as little as possible with each other.

Figure 6.7 shows an example of two such textures using mirrored hatch patterns (A, B) to encode two attributes. Pattern A encodes the presence of a given word in the comment message associated with a version, and pattern B encodes the author of that version. Figure 6.7 shows the evolution of four files across two versions. Color encodes file type. One can easily see that file F_3 has only attribute values encoded by pattern A, and file F_4 only attribute values encoded by pattern B. File F_1 has values encoded by both patterns, since drawn with the crosshatch combination of patterns A and B. File F_2 has none of the two attributes, *i.e.*, is not committed by the sought author, nor does it contain the sought word, as it shows no texture.

Further analysis of Figure 6.7 shows more correlations. Pattern B appears in both version V_1 and V_2 of F_4 , so F_4 was committed by the same author twice. Pattern A has different values for versions V_1 and V_2 of F_3 , so different words of the searched set appear in them. File F_1 is committed by the selected author (has pattern B), and contains different searched words in its two versions. Comparing F_1 with F_3 , we see that the search hits are the same in the respective versions of the two files. Version V_1 of F_1 is more similar to the a_1 value of pattern A than to the a_2 value. Hence, one could conclude that version V_1 of F_1 contains the word a_1 and is committed by author b_1 , and version V_2 contains the word a_2 and is committed by the same author b_1 .

Figure 6.8 shows a second example of visualizing several attributes. Here, we use bubble patterns to indicate revisions belonging to a given system release, and a diagonal hatch pattern for files containing the word *tag* in their commit logs. Color shows author ID. We can easily spot files belonging to the selected release and containing the word *tag*.

Preliminary user studies show that superimposing textures obtained by scaling perceptually largely different patterns can encode two, sometimes three, attributes simulta-

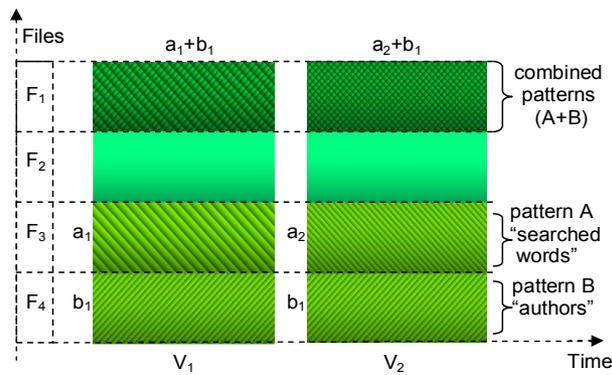


Figure 6.7: Combining texture patterns to show several attribute values

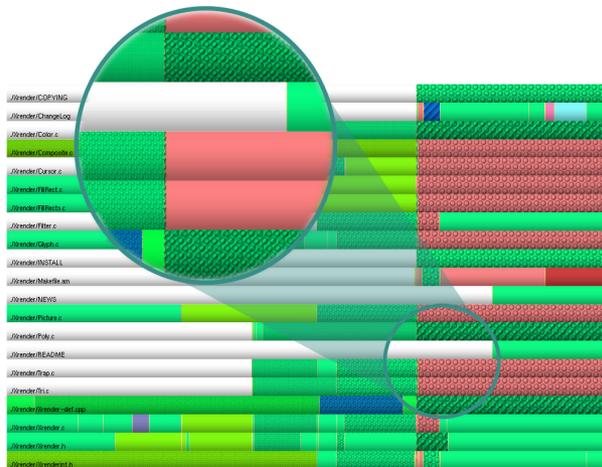


Figure 6.8: Texture composition: spheres = selected revisions, hatches = word 'tag' in comment

neously. The most effective use hereof is for showing nominal attributes with a small value range, such as file types, search hits from a small word set, or author IDs. Indeed, superimposing textures, even when carefully chosen, decreases the individual pattern resolution, which makes it quite hard to map continuous values with high precision. After experimenting with several patterns, we designed a small set containing vertical, horizontal, and the two diagonal hatches, and also a “bubble” pattern. This set is quite effective since the interference between any two patterns in this set is quite small, and the patterns are easily distinguishable, even when drawn on small areas and/or scaled to small resolutions.

Yet, there is a limit to how small an area we can texture and still see the patterns. This minimal size seems to be around 25×25 pixels on a normal 19 inch screen.

Navigation in Viewpoint Space

The second challenge we faced when building our multivariate visualization was to find an intuitive and easy way to define, customize, and navigate between different evolution views. We describe next our approach to this challenge.

First, we defined a *view* as a function

$$f_i : Files \times Versions \rightarrow Color$$

$$f_i(V_{kl}) = rgba$$

that associates a color $rgba \in Color$ to every file version V_{kl} based on its attributes. Then we used the preset controller mechanism proposed by Van Wijk and Van Overveld in [112] to switch between and correlate the views. This mechanism works as follows. Given some 2D points p_i that correspond to the views f_i , and an “observer point” p , the user can define custom views f_c

$$f_c(p, \{p_i\}, V_{kl}) = \frac{\sum_i (d(p, p_i) \cdot f_i(V_{kl}))}{\sum_i d(p, p_i)}$$

where $d()$ is some inverse distance function between points, *e.g.*, $d(x) = \frac{1}{(1+x^2)}$. The observer p and the view points p_i are identified by glyphs centered on their position in the 2D space. The custom views are generated by moving either p or p_i associated glyphs with the mouse in the preset controller widget.

Next, we refined this mechanism to make it more effective for software evolution visualization, as follows. To give users better feedback about the way each view influences the final image, isolines are drawn around the observer glyph. This helps measuring the observer-view distance and hence estimate the “strength” of a given view. We saw that this matches closely the way users build visualizations: it is not important to specify the exact contribution of one view in the final visualization, but rather to indicate the relative contribution of all involved views.

A second addition we made was to use glyphs parameterized by *view* attributes. The purpose of these glyphs is to form some intuitive metaphor that suggests what kind of visual mapping the preset associated with that glyph does. Consequently, this should give the user a hint about what to expect when moving the controller towards that glyph. To this end, we applied design principles validated in the gaming industry by products such as Microsoft’s Age of Empires [2], where various attributes (*e.g.*, offence, defence, quality, and stamina values of soldier figures) are drawn on a small screen area with a few colors.

Figure 6.9 illustrates our solution on a preset controller scenario having seven possible views. Only two views contribute to the visualization, *i.e.*, *authors* and *search text*, as the other views are beyond the furthest observer isoline. The *authors* view encodes every version segment in the evolution visualization with a color depending on the ID of the author that committed the version. The associated *authors* glyph contains a number of colored squares, one per user, showing the users’ colors. The *search text* view encodes the file versions that contain a given string in their associated commit comment with a string dependant color. Several strings can be searched for at the same time. Each string

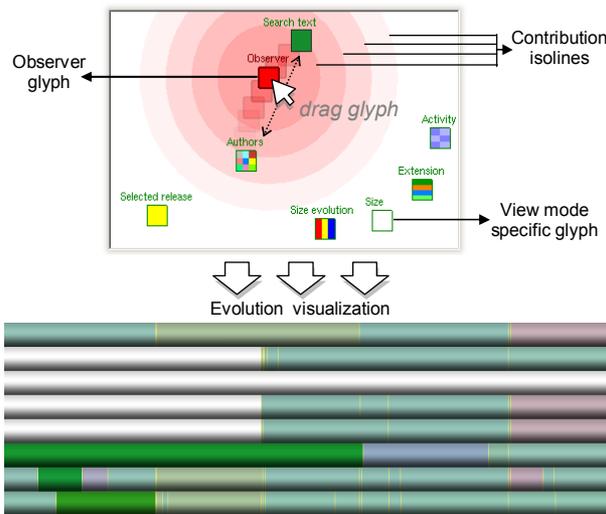


Figure 6.9: Preset controller based navigation among possible evolution views

has an associated color. Versions that contain several strings are colored with a special color (red). Versions that do not contain any of the searched strings are grayed out. The associated glyph of the *search text* mode shows a vertical strip for every string in the search list, colored with the string's color. As the observer is closer to the *search text* glyph, the final visualization will be mainly influenced by this view. The glyph associated with *search text* has only one color, *i.e.*, green, so we search for only one string. Hence, the search hits will appear as saturated green in the resulting evolution visualization. The second active visualization mode, *i.e.*, *authors*, has a smaller effect as its glyph is further away from the observer. Hence, the authors' colors will be less saturated, yet visible enough to distinguish between different authors or identify specific ones. The *authors* glyph contains a large number of colored squares indicating the user should expect a large number of authors to show up.

In general, we designed the glyphs as small treemap-like areas with cells that show the colors the mapping f_i of that respective glyph can generate (Figure 6.10). Clearly, this approach works well only if the cardinality of *Colors* is small (*i.e.*, under 15).

Informal user studies that we have organized indicate that our modified preset controller is a very intuitive and fast way to understand and create the attribute mapping used in our visualization. Although only one attribute can be mapped by color at a given time, cross-view correlations are still possible. They are enabled by the seamless and fast transition between different views. By repeatedly shifting the observer's position between several views, one can correlate the color determined by the current predominant view with the previous color, stored in the short term memory. Seamless transition between colors by means of blending helps focusing user's attention on an area of interest, as one is less distracted by sudden changes in other parts. Conversely, the repeated shifting of the observer glyph helps refreshing the short term memory.

A similar approach can be investigated for views that use texture to encode attributes.

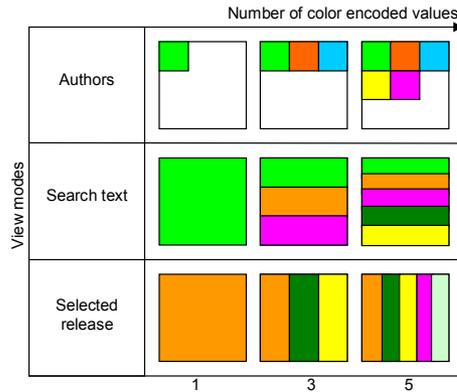


Figure 6.10: Parameterized glyphs for view mode identification

Next, the challenge would be to build a controller that enables both switching between views and combining them, such that color and texture can be used independently at the same time. A solution could be to use more preset controllers: one for each visual dimension that could be separately identified in the image. For instance, in a visualization where two attributes are displayed at the same time, one with color and the other with texture, two preset controllers can be used. Another alternative could be to use multiple observers points in the same preset controller, addressing different sets of views.

6.3.4 Multiscale Visualization

The clustering-based layout mechanism proposed in Section 6.3.1 uses an agglomerative bottom-up algorithm to produce a binary decomposition tree of a system. The tree leafs are all files $\{F_i | i = 1, \dots, NF \in \mathbb{N}\}$ in the project and its nodes are the computed clusters. We denote the *file-set* of a node by $T(n)$, *i.e.*, the set of leafs which are descendants of $T(n)$. A decomposition of the system is a set of nodes N_{sys} that has the properties:

- $\bigcap_{n \in N_{sys}} T(n) = \emptyset;$
- $\bigcup_{n \in N_{sys}} T(n) = \{F_i | i = 1, \dots, NF \in \mathbb{N}\}.$

In other words, N_{sys} is a partition of $\{F_i | i = 1, \dots, NF \in \mathbb{N}\}$. Once the decomposition tree is computed, the next question is: “How do we let users construct and select meaningful decompositions?”

Decomposition Selection Methods

A straightforward mechanism for selecting a decomposition N_{sys} is to include all roots of the intermediate cluster trees that are present in the k^{th} step of the clustering algorithm.

In the case of the bottom-up agglomerative clustering, this leads to a decomposition with $NF - k$ clusters, where NF is the total number of files (see Figure 6.11a). This mechanism is useful for implementing scenarios of the type “display the project as a set of n similar components”. We call this the *step-based* method. However, specifying a decomposition via the step-based method can lead to the coexistence of clusters containing children that are highly related with clusters containing less related children, according to the clustering criteria (see *e.g.*, Figure 6.18a). This makes the clusters difficult to compare, and consequently, the decomposition is difficult to understand by the user.

To deal with this issue of the step-based method, we provide an approach for selecting a cluster decomposition N_{sys} based on node properties and not creation order. In this approach every cluster node n gets a relevance factor $R(n)$. The tree is traversed in pre-order, and at each step the relevance of the current node $R(n)$ is compared to a user-selected value R . If $R(n) > R$, n is added to the selection and its children are skipped, else traversal continues with the children of n . In the resulting decomposition N_{sys} , most clusters have a similar, though not guaranteed equal relevance with the reference value.

One value we tried for estimating the cluster relevance was the cardinality of the file-set associated with a node. This alternative produces decompositions with clusters of similar size, containing files that are related to some extent. However, the purpose of this decomposition is purely to reduce the visual complexity of the data representation, and no decomposition meaning can be associated with the clusters. Due to the lack of useful usage scenarios we dismissed this alternative.

A second alternative we used for calculating the node relevance was the cluster cohesion (Figure 6.11b). We found this approach useful as it can be used to divide the system in components of the same type, depending on the clustering criterium and cohesion func-

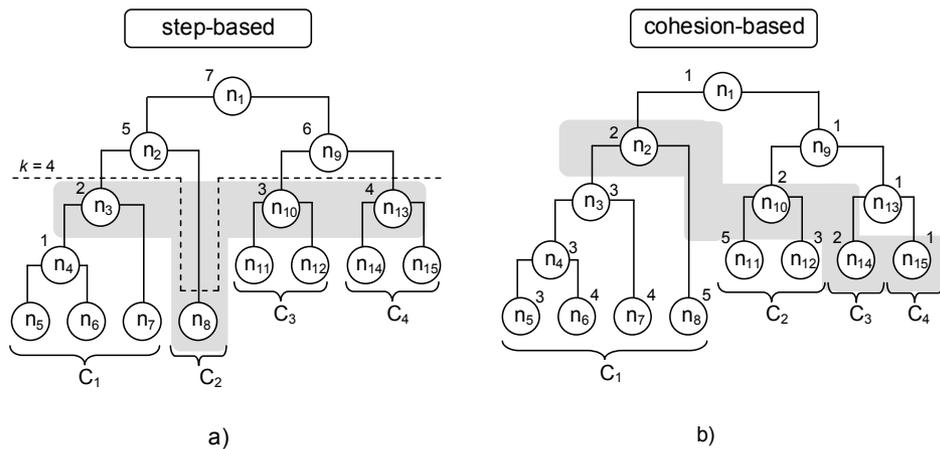


Figure 6.11: Decomposition selection methods – gray regions show selected nodes: (a) *step-based* – numbers next to nodes indicate the clustering step that produced the corresponding cluster; the highlighted decomposition is for the 4th step; (b) *cohesion-based* – numbers next to nodes give node cohesion (larger means higher); the highlighted decomposition is for cohesion 2.

tion. We call this the *cohesion-based* cluster decomposition selection.

Cluster cohesion can be computed based on the cluster diameter, *i.e.*, the distance between its two children. This leads to a decomposition selection mechanism similar to the one proposed by Seo and Shneiderman in [95]. A drawback of this approach is that the cohesion of the children does not always propagate to their parents. Consequently, one can make no distinction between nodes that have the same cohesion but are based on children of contrasting relevances. A carefully designed distance metric and cluster merging criterion can take care of this problem. Another approach to ensure cohesion inheritance is to compute node cohesion as the size-weighted average of the children cohesions, biased with the distance between the children. In this way children relevance propagates to their parents, yet nodes are less relevant when the distance between children is large, compared to cases when the distance is small. The node cohesion $C(n)$ can be computed recursively using:

$$C(n) = B(d_n) \cdot \frac{C(n_{c1}) \cdot |T(n_{c1})| + C(n_{c2}) \cdot |T(n_{c2})|}{|T(n_{c1})| + |T(n_{c2})|}$$

where d_n is the diameter of the current cluster n , n_{c1} and n_{c2} are its two children, $|T(n_x)|$ is the size of a node file-set, and $B(d_n) = \frac{1}{(1+d_n)^k}$, with k constant $\in \mathbb{N}$, is a bias factor that depends on the diameter. The leaf nodes cohesion is considered to be 1.

We applied both the step-based and the cohesion-based decomposition selection methods presented above only on the decomposition tree given by the bottom-up agglomerative clustering algorithm discussed in Section 6.3.1. However, it is clear these methods can be applied on a tree resulting from any clustering algorithm.

Navigation in Decomposition Space

Another issue of selecting a decomposition is the missing link between the decomposition selection mechanism and the desired level of detail (LOD). The step-based decomposition selection requires the user to select a desired number of clusters. However, it does not indicate how relevant the selected clusters are. The cohesion-based method allows the user to specify a desired cohesion level and provides a decomposition that tries to closely match that level. However, the user still has to guess a “good” value for the cohesion, such that the decompositions is comprehensible. In practice, we saw that users needed to continuously adjust the input parameter until a compromise is reached between cohesion and cluster size and number. To assist the user in making a good choice, we propose a new visualization: the *cluster map*. This combines a classical value-selecting slider with a 2D map in dendrogram style of all the available decompositions (Figure 6.12). The horizontal axis maps the LOD (number of clusters for the step-based method, or the cohesion value for the cohesion-based method). The vertical axis depicts the cluster decomposition for every value on the horizontal axis. Every cluster decomposition is drawn as a vertical stack of cushioned rectangles, all stacks having the same width. The height of each rectangle is proportional with the number of files contained in the associated cluster. Intuitively, each stack in this visualization is actually a mini-map of the plateau cushions used in the main visualization (*e.g.*, Figure 6.18) to show the complete system decomposition.

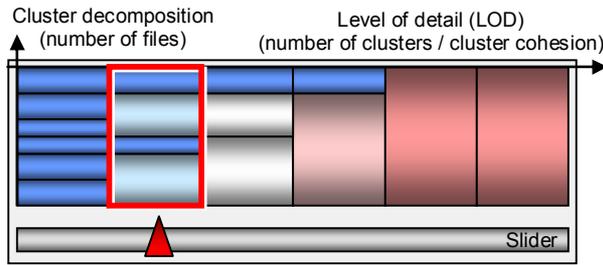


Figure 6.12: The cluster map widget. Clusters are drawn as cushioned rectangles. Color encodes cohesion.

A blue-white-red colormap encodes the cohesion (low to high) of each cluster drawn as a rectangle.

The widget (shown in Figure 6.12) enables users to quickly identify and make a compromise between the desired cluster size and cluster cohesion. Also, it enables the user to select the desired decomposition via the slider at the bottom. Furthermore, users can correlate the clusters depicted in the main evolution visualization with the ones in the widget and therefore identify their cohesion.

However, drawing all clusters of typical software decompositions in the cluster map leads to aliasing problems, as the cluster cushions easily become less than one pixel high. This creates the false impression that there are no clusters on the finer levels of the cluster map, *e.g.*, at the left of Figure 6.13a. We solve this by drawing, on every level of the cluster map, only those clusters whose screen height exceeds 3 pixels, since this is the minimal height at which cushion textures are distinguishable. As shown in Figure 6.13b, small clusters are now clearly visible. An added bonus of this is that rendering the cluster map takes now constant time for arbitrarily large hierarchies.

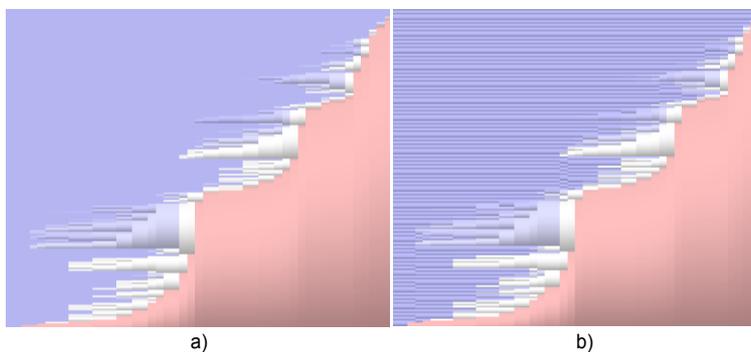


Figure 6.13: Cluster map: (a) without antialiasing; (b) with antialiasing.

Figure 6.14 shows an example of 2D cluster decomposition maps corresponding to two presented selection methods, for a project with 28 files spanning across up to 21 versions. All horizontal axes are normalized and sampled with a rate of 1:25, *i.e.*, the horizontal axis displays and allows the selection of 25 values uniformly distributed between

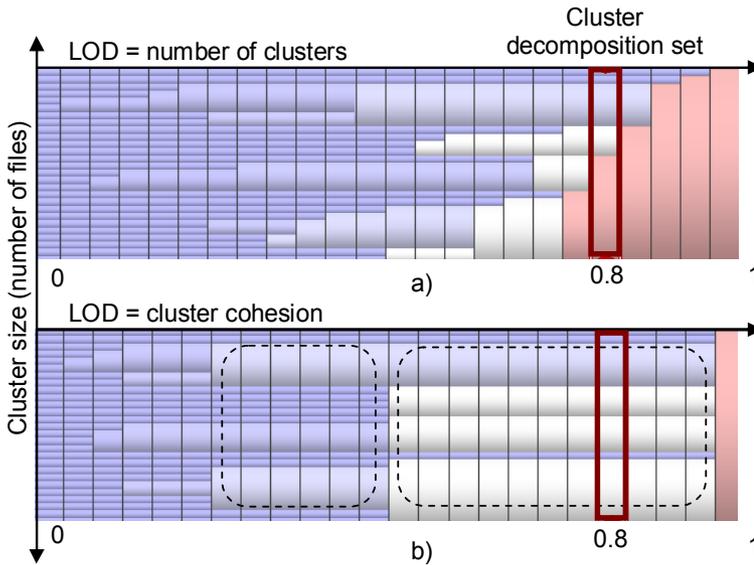


Figure 6.14: 2D cluster map using: (a) step-based selection; (b) cohesion-based selection. Clusters are drawn as cushioned rectangles. Color shows cohesion. Dashed rectangles show intervals with the same cluster decomposition.

0 and 1.

Figure 6.14a depicts the cluster map for the step-based method. The horizontal axis gives the normalized level of detail: 0 is for 100% detail, *i.e.*, every file is a cluster, 1 is for 0% detail, *i.e.*, the whole project is seen as one cluster. We can now easily see that at some levels clusters have very different cohesions. Consequently, we cannot easily compare them, and associate a meaning to the decomposition. Therefore, for this particular project and cohesion measure, the step-based decomposition selection can be misleading or difficult to understand.

In Figure 6.14b, the cohesion-based selection method is illustrated. The horizontal axis gives the normalized cohesion: 0 is for maximum cohesion, 1 for minimum. We can see that at most levels of detail clusters have similar cohesion. In practice, these may correspond to structural or logical building blocks for the system. For instance, depending on the cohesion level, the clusters of one level can give a system decomposition in meaningful structural components, such as classes or packages. Additionally, large clusters at a given level of detail may signal the presence of complex software components. This information may be very useful when trying to understand the software system.

In the case of the cohesion-based method presented in Figure 6.14b, the cluster decomposition set does not vary for every LOD value on the x axis. There are large LOD intervals that have the same set (see dashed rectangles). In general, such long “constant” intervals border an important system decomposition step in terms of cluster cohesion. Consequently, a carefully designed cohesion factor can show passing from highly coupled system components, *e.g.*, classes, to more loosely coupled ones, *e.g.*, packages, and

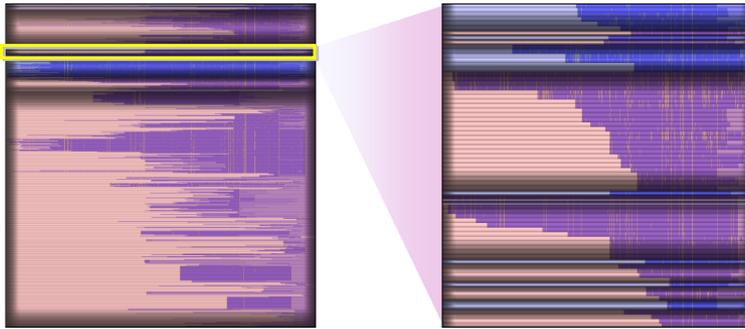


Figure 6.15: Zoom adjusted cushion height for visual segregation

therefore can give an insightful, intuitive, and simple structural view on the system.

It is true that our cluster map alone cannot show which are the meaningful partitions to visualize for a given system and problem. However, it shows which are those values of the level of detail parameter where relatively important clustering events, *i.e.*, system simplifications, take place. The user can decide to select these levels and visualize the corresponding decompositions, without having to browse all the (usually quite many) level of detail values. The cluster size distribution in the cluster map shows what kind of visualization to expect if selecting that level of detail.

6.3.5 User Interaction

To validate the techniques that we propose in this section, we implemented them in CVSgrab, a tool for visual mining of CVS repositories. CVSgrab gives an intuitive 2D overview on the evolution of complete projects at file level. The tool facilitates exploratory layout building and correlation making by providing a rich interaction palette, following Shneiderman's guidelines [96]: overview first, zoom and filter, then details-on-demand.

Industry-size projects may contain thousands of files whose history spreads across more than one decade. To facilitate access to details, CVSgrab provides zoom and pan facilities. Zoom presets enable easy access to standard view modes, *e.g.*, fit image to screen, fit file to line size. Some visual elements have a zoom-adaptive behavior to preserve their visual efficiency across different levels of detail. The plateau cushions, for example, have a zoom-dependant height such that their appearance remains the same in the border regions. In this way, the visual segregation of clusters becomes independent on the zoom level at which it is performed. Figure 6.15 illustrates this. The right image shows a 20-fold magnified inset of the data shown in the left image. Still, the cushions shown in the right image look similar to the ones in the left one.

CVSgrab implements also a details-on-demand mechanism that enables users to get detailed information about a selected or mouse-brushed version. Information such as precise size, file name, and author comments logged at commit time, is displayed in textual format in a separate view.

The following section presents the results of a number of evolution assessments we performed on industry-size projects. These illustrate how the interaction mechanisms presented above and the visualization techniques described in Section 6.3 can be successfully used to investigate the evolution of real life software systems.

6.4 Use-Cases and Validation

We analyzed the use of CVSgrab for mining the history of several industry-size projects. Here we present the results of such investigations for two projects: VTK [118] and MagnaView [74]. VTK is an open source project of over 2700 files written by 40 developers in over 11 years. MagnaView is a commercial visualization software package containing 312 files, written by 11 developers in over 16 development months.

6.4.1 Insight with Dynamic Layouts

To validate the efficiency of the layout and mapping mechanisms of CVSgrab, we performed an informal user study. In this study, the VTK project was mined by three experienced C++ developers having, however, no VTK knowledge. They participated first in a 15-minute training in which the functionality of CVSgrab was explained on a small example project, with several generic use cases that could be easily reproduced on other input data. Next, they mined the history of VTK for 2 hours. Finally, their findings were assessed by a developer with over eight years of VTK experience.

Figure 6.16 depicts various annotated visualizations of the complete project evolution obtained during the study using sort operations. In Figure 6.16a, 6.16c, and 6.16d files are colored on activity, as detailed in Section 6.3.1. Yellow lines show commit moments. In Figure 6.16b files are colored on author *ID*, every hue encoding an author. While this might create confusion when establishing the identity of users encoded by similar hue, it gives a good overview of major overall patterns.

In Figure 6.16a files are sorted alphabetically. Although cluster cushions are not rendered, a vertical metric view (C) shows the clusters to which files belong, using color mapping. The alphabetical sorting of files uses the full pathname and thus nicely groups together files in the same folders. By mouse brushing the evolution area, the users easily identified the major folders of the project, highlighted in (A): *Imaging*, *Graphics*, *Contrib*, and *Common*. The names were made available as details-on-demand in the visualization window's status bar. Two compact, low-activity evolution regions were also spotted (B). By brushing the corresponding evolution area, the users discovered, via the status bar information, that they refer to VTK code examples in Python. The vertical metric view (C) is fragmented in the color space. This helped the users conclude that the project's organization based on evolution coupling (*i.e.*, cluster decomposition) does not correspond entirely to its organization as a set of folders.

Sorting on creation time allowed the users to find several possible moments of so-called punctuated evolution (E), *i.e.*, moments when large code changes took place in a short time. The details-on-demand feature helped refining their hypotheses about these

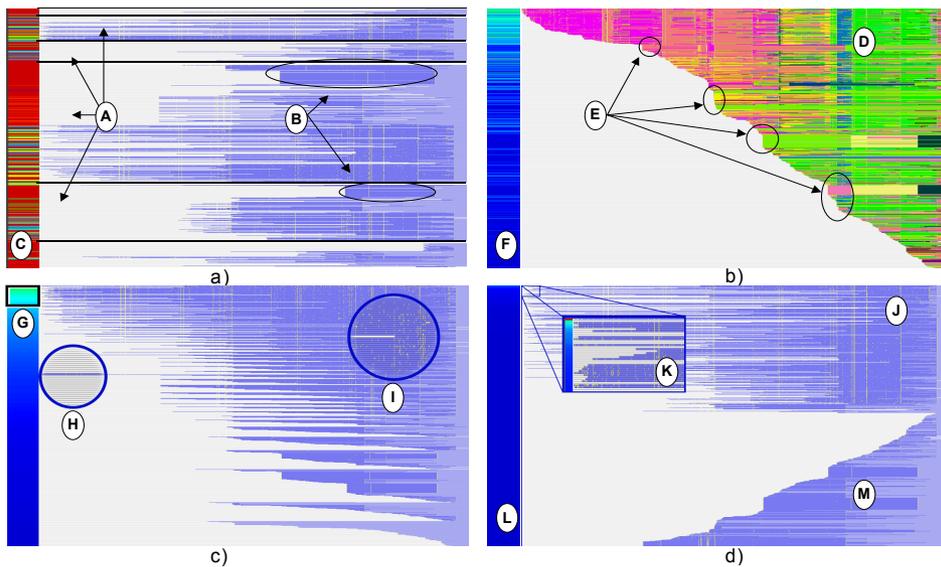


Figure 6.16: Interactively built layouts of the VTK project using sort operations: (a) files sorted alphabetically, vertical metric shows cluster IDs; (b) files sorted by creation time, vertical metric shows activity; (c) files sorted by activity, vertical metric shows activity; (d) files sorted by evolutionary coupling measure with respect to a reference file: `vtkIntArray.cxx`, vertical metric shows evolutionary coupling.

events. Of the four moments highlighted in the image (E), three refer to the addition of VTK examples, and just one involves heavy changes of the library functionality.

Further, as visible in the image, the vertical metric (F) has no smooth transitions. This made the users assume there is no direct correlation between creation time and file activity. Indeed, the project contains both files that were introduced early but recorded little activity, *e.g.*, stable interfaces and/or implementations, and files that were introduced later but were frequently updated, *e.g.*, problematic and/or unstable implementations.

In Figure 6.16c files are ordered according to their recorded activity. The vertical metric view (G) depicts also the activity measure using a rainbow color map (red = high activity, blue = low activity). From this image, the users concluded that most development is concentrated in less than 10% of all files (G), with a few files, *e.g.* `vtkRender.cxx`, `vtkPolyData.cxx`, `vtkImageData.cxx` being frequently updated. Indeed, these files contain fundamental, core-related structures of the library. Figure 6.16c was also useful to find the activity outliers. The highlighted inset (H) depicts an example of an early outlier, *i.e.*, a stable file during evolution: `vtkRender.h`. The highlighted inset (I) depicts a late outlier, *i.e.*, a file introduced later, but often updated: `vtkDataObject.cxx`.

Finally, in Figure 6.16d, files are arranged according to their evolution similarity (*i.e.*, measured by the evolutionary coupling metric) with respect to a selected reference file: `vtkIntArray.cxx`. The vertical metric view (L) uses a rainbow colormap to depict the evolutionary coupling measure (red = very similar; blue = very different). The users

concluded that the chosen reference file had little in common with most of the other files in the project, as the metric view is almost entirely blue. In the magnification caption (K) a zoomed-in region of the evolution area (J) is displayed. This revealed a small number of files that had a higher evolutionary coupling value. Via the details-on-demand mechanism the users discovered their identity: `vtkLongArray.cxx`, `vtkFloatArray.cxx`, `vtkBitArray.cxx`, etc. Indeed, detailed inspection confirmed these files have a tightly coupled implementation. The files depicted in region (M) are arranged in decreasing order of their creation time. They represent actually files that have no evolution similarity with the reference one and are sorted according to a secondary criterion.

At the end of this study, we summarized the three users' observations and checked them again with the knowledge of the expert developer. The expert validated the largest part of the observations as fully correct. One aspect he found himself novel was the lower-than-expected number of files from the project core, *i.e.*, files where most of the activity is concentrated (see G in Figure 6.16c).

6.4.2 Complex Queries

The texture-based attribute encoding of CVSgrab lets users visualize up to four attribute values at the same time (three textures and one color). This supports complex evolution queries. The *preset controller* takes the correlation possibilities one step further. Figure 6.17 addresses an example of complex query applied on the evolution of MagnaView:

“What versions of GUI specification files, belonging to release 549, and containing the word bug in the associated log message, have been committed by developer tomasz?”

We answered this query with the following techniques:

- a diagonal hatch pattern texture in the direction NE-SW to show versions containing the word bug their commit message
- a diagonal hatch pattern texture in the direction NW-SE to show versions that belong to release 549
- an author ID-to-color view mode, with red encoding *tomasz*
- a filetype-to-color view mode, with gray for GUI specification files
- a preset controller to switch between the two color view modes

Figure 6.17a depicts a zoomed-in area of the evolution visualization using the author ID view mode. The highlighted versions are possible candidates for the query above. The cross-hatch texture pattern shows they both contain the text “bug” and belong to release 549. Moreover, red indicates the versions have been committed by *tomasz*. Using the preset controller to rapidly change between the two view modes, one can see that only one of the candidate versions is a GUI specification file: `UEditViewForm` (highlighted in Figure 6.17b).

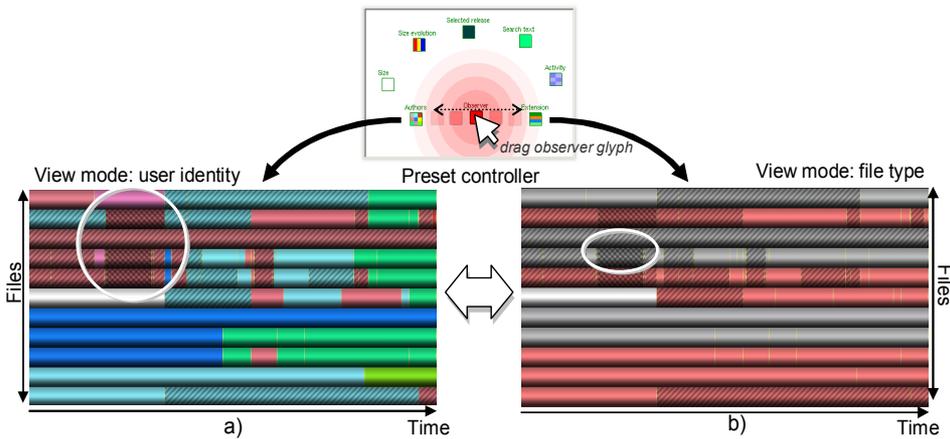


Figure 6.17: Complex queries usage scenario. Blended textures and colors show a set of possible solutions based on three attributes. Using the preset controller, a fourth attribute can be checked and the set of possible solutions (a) is reduced to one version (b).

Many other similar scenarios and use cases exist. Using the proposed multivariate visualization features, one can easily give answers to complex queries by narrowing down a set of candidate solutions using a visual approach.

An alternative to this is to use a data analysis approach for identifying the result of the query first. Subsequently only the result is to be visualized, for instance using a two color encoding: red for versions that are in the result, grey for the rest. This makes the versions that satisfy the query imposed criterium very easy to identify. Nevertheless, the disadvantage of this approach is that no information is given about the composition of the result, *i.e.*, the versions matching the criteria imposed by the composing queries. Questions such as "are all versions satisfying the first condition of the query also complying with the second one?" cannot be addressed. The first alternative supports finding correlations in this direction, and therefore it offers more insight.

6.4.3 System Decomposition

The cluster map widget allows users to interactively select a partition of the system evolution at file level. Figure 6.18 shows the use of the cluster map widget in combination with the two presented partition selection mechanisms, *i.e.*, step-based and cohesion-based, and presents decomposition results for the VTK graphics library. The left part of the figure shows the cluster maps for the project evolution for the step-based (a) and the cohesion-based selection method (b). Both widgets use a red-to-blue gradient color map to show (low to high) cluster cohesion. In each widget, the chosen selection is indicated by a red rectangle. The right part of the image depicts the results of the chosen cluster selection in the main evolution visualization, *i.e.*, clusters are drawn as plateau cushions over their respective files.

For the chosen LOD, the step-based selection method produces a decomposition con-

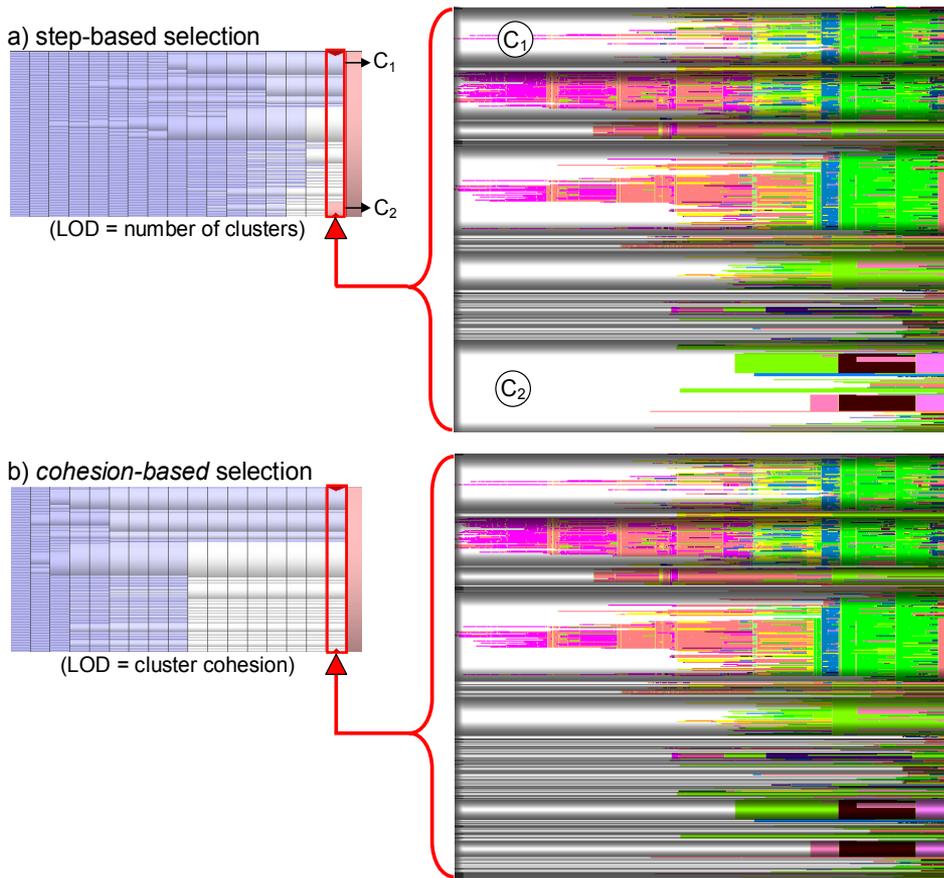


Figure 6.18: Cluster decomposition selection scenario

taining clusters of very different cohesions (see Figure 6.18a). Clusters C1 and C2, for instance, are at the opposite ends of the cohesion range. Consequently, it is difficult to compare them and to assign a meaning to the evolution decomposition depicted in the right part of the image.

Figure 6.18b shows a system decomposition using the cohesion-based method. The right part of the image, depicting the evolution decomposition for the selected LOD, is somewhat more cluttered than in the previous case (*i.e.* it contains more clusters even for high level decompositions). However, as shown by the associated cluster map widget, all selected clusters have a similar cohesion. Consequently, they are easier to compare. Additionally, the cohesion-based selection contains also large intervals of constant decomposition. They are caused by a project specific number of cohesion thresholds. On the several projects we checked this method in practice, these intervals corresponded to meaningful structural decomposition views on the system.

In conclusion, the cohesion-based selection method can be a better alternative to the step-based one. It generates similar-cohesion cluster decompositions. This is visible in

the cluster map widget. The role of this widget is threefold. First, it shows to the user a global picture of the system decomposition, thereby letting one assess the quality and meaningfulness of a decomposition method. Secondly, it shows constant intervals of the decomposition, which very often correspond one-to-one to different system structurings. Thirdly, it is an useful instrument to compare the quality of various decomposition methods and see the effect of tuning the clustering metrics.

6.5 Conclusions

In this chapter we have presented a set of visualization and interaction techniques that support history mining of large-scale software projects at file level.

We first proposed a novel technique for layout of file evolution representations, by interactively mixing and adjusting sort and cluster operations to direct the visual mining towards specific goals. We used horizontal and vertical metric views to enable evolution correlations based on more sort criteria at the same time.

To enhance the correlation capabilities, we adapted and extended two existing techniques that enable the visualization of more attributes at the same time at the same screen location. With the new additions, up to four attributes can be encoded at the same time using color and hand-designed texture patterns, minimizing the visual interference. Color-based correlations across several color-encoded attributes are enabled using an extended preset controller technique.

Subsequently, to reduce the visualization complexity, we introduced a simple-to-use, yet powerful clustering technique. This technique reduces the project visualization to a smaller number of clusters with files having similar evolutions. The typical tasks targeted with this approach are of the kind: “show the whole project split into n similar components”. We reduced the interference between the cluster rendering and file colors using a mixed cluster luminance and hue encoding. This combines the visual comfort of hue-based cluster segregation with the precision of the plateau cushions in the boundary regions.

We have also presented and compared two methods for selecting cluster decompositions from a hierarchical decomposition tree (*i.e.*, step-based and cohesion-based). The step-based method allows the user to specify exactly the number of components in which the system is to be decomposed. The cohesion-based method generates decompositions in which clusters have similar cohesion. We enabled users to see an entire decomposition tree and select a meaningful level from it using a new widget: the cluster map. This widget enables users to quickly assess the results of a selection method in the context of a specific project, and choose the cluster decomposition that matches some desired compromise between level of detail and cohesion.

We validated the proposed techniques by implementing them in CVSgrab, a visual tool for exploring the evolution of industry-size projects. The dense pixel visualization combined with multivariate attribute encoding and interactively built layouts makes it possible to navigate and assess code projects beyond the size of what is possible by similar tools [69] or with better insight [125]. For example, one can get a comprehensive

overview of the complete evolution of the VTK project (2700 files, 40 developers, over 11 years, about 100 versions for active files) with quite little interaction.

CVSgrab does not allow visualizing code at line level. For this, other tools, such as CVSScan described in Chapter 5 are best used. CVSgrab's main strength comes when one does not know where (and why) to zoom in, given a large software project of many versions. Additionally, the evolutionary coupling based sorting and clustering can be effectively used to discover relations between files in a project that are not apparent, without needing to use more the complex, slower, language-specific parsing of the files' contents.

Chapter 7

Visualizing Software Evolution at System Level

Many software evolution assessments are initiated and performed at system level. The goal of this type of assessments is to understand major evolution trends in the quality of the software system and possibly trigger more detailed investigations. To this end, emphasis is put on analyzing and discovering correlations between evolution trends, and the system is observed at a high level of abstraction. In this chapter we propose a customizable visualization that enables users to analyze the evolution of a wide range of software related metrics at system level. Central to this visualization is a chart based image and an easy way of customizing it for a specific analysis. Data and visual sampling issues are addressed such that information is presented in an unambiguous and meaningful manner. The suitability of the proposed approach is demonstrated with a number of analysis experiments that we performed on existing real-life systems.

7.1 Introduction

Comprehensive software quality assessments are performed by skilled professionals and they may require substantial resources. However, assessments of this type are often triggered by quick analyses of the system at a high level of abstraction. Additionally, high level analyses are also useful during detailed assessment, to direct investigations and to obtain an overview of the problem at hand.

In this chapter we propose a visualization that enables users to investigate the evolution of a system at a high level of abstraction, by revealing overall system quality indicators and evolution trends. Typical questions we try to answer with this are:

- How does the system complexity evolve?
- How difficult is it to maintain the system?

- How much effort is required to maintain the system using the current human resources?
- How is system knowledge distributed over the team?

The structure of this chapter is as follows. Section 7.2 presents the type of attributes and evolution related assessments that our visualization addresses. Data sampling issues are also considered, such that only meaningful information is presented to users. Section 7.3 details a number of alternative visual encodings that can be used to present the information and gives usage guidelines. It also addresses the multivariability and the visual scalability issues, such that information interpretation can be done in a meaningful way. Section 7.4 illustrates the applicability of the visualization that we propose with a number of experiments performed on industry-size real-life projects. Section 7.5 concludes the chapter.

7.2 Data Model

One of the most commonly used ways to assess the quality of a system is by measuring a number of content based quality indicators called software metrics, which are concrete implementations of the version metrics introduced in Section 4.4.2. Comprehensive overviews of the most commonly used software metrics are given in [43] and [65]. The examples presented in this chapter illustrate only some of them:

- the software system size in lines of code;
- the cyclomatic complexity of a software system [79] (*i.e.*, the number of branching statements, such as `if-then-else` or `case` statements);
- the number of files containing contributions from a given author.

Software metrics are of two major types: categorical and numerical [80]. Numerical metrics express the quality of a software entity as a number, which can be easily aggregated in overall summaries (*e.g.*, size of a system = sum of the sizes of its files). Categorical metrics indicate whether the targeted software entity belongs to a given category (*e.g.*, file *A* belongs to the files developed by user *X*). Given a set of software entities and a metric that produces categorical values for those entities, overall summaries can be constructed by counting the number of entities that belong to a specific category according to the metric (*e.g.*, number of files developed by a given user). Numerical metrics can be transformed into categorical metrics by dividing the value domain in intervals (*e.g.*, size of a file belongs to the interval 1-10 lines of code). In this way, categorical summaries can be obtained also for numerical metrics. This can be useful when presenting results to a nontechnical / managerial audience. On the one hand this type of users are more familiar with quantity, percentage and category based assessments. On the other hand, the translation process from a value domain to a set of intervals often requires a labeling of the intervals and, implicitly, a semantic interpretation of the results (*e.g.*, file size in interval 1-100 lines of code = “file is easy to understand”). In this way, the technical findings that the assessment produces are brought closer to the nontechnical audience.

Both numerical and categorical metric summaries are useful for getting insight in the state of a system at a single given moment. While the figures can be interpreted from a generic perspective or set of best practices, this approach does not take into account the specific development context of a project. Better assessments can be made by getting insight in this context via system level evolution assessments, and interpreting current findings from the perspective of previous situations.

System level software evolution assessments are performed on a particular implementation of the software evolution model presented in Section 3.3. The system is regarded as a single entity, and its evolution is followed in time. Consequently, the entity similarity functions Γ (see Definition 3.2.1) are defined on sets with one element that represents the system entity itself at a specific moment:

$$\Gamma_S : \{S\}^i \times \{S\}^j \rightarrow [0, 1]$$

It is, therefore, less interesting to identify the evolution patterns described in Section 3.2 (*i.e.*, only one pattern exists: continuation). The emphasis is in this case entirely on investigating trends of the system attributes and discovering correlations between them. System metric summaries are practical implementations of such attributes, commonly used by the software engineering community.

Consequently, the data involved in the visualization of software evolution at system level can be modeled as a set Δ_S of sequences M^i of numerical and categorical metric values

$$\Delta_S = \{M^i | i = 1, \dots, N \in \mathbb{N}\}$$

where N is the number of sequences. These sequences have the same cardinality K , equal to the number of revisions the system had on the investigated period.

$$M^i = \{(M_j^i, t_j) | j = 1, \dots, K \in \mathbb{N}\}$$

where K is the common sequence length and $t_u < t_v$ if $u < v$. The position of each element M_j^i in a sequence is associated with a time stamp t_j . Sequences are sorted in the increasing order of the time stamps. Elements at the same position in different sequences have the same associated stamps. Consequently, the element position can be used to perform time based correlations across sequences.

7.2.1 Data Sampling

The evolution of a software project can cover many years. The history data extracted from SCM systems is reported with an accuracy of one second, expressed as an integer number starting from 1st of January, 1970. Given the purpose of software evolution visualization at system level, presenting the entire data with accuracy in terms of seconds might not be important to the user, and values on a coarser time scale are preferred [113]. Consequently, the first issue of software evolution visualization at system level is how to sample data on the time axis such that meaningful metric summaries can be presented to the user.

The straightforward solution is to divide the time axis in intervals and derive an av-

erage of the metric values corresponding to data recorded in each interval. User-defined intervals of equal length (in seconds) are not appropriate, given the uneven division of years and months in days. Additionally they enable the user to explore data at unfamiliar levels of detail, which could lead to wrong assessments. Consequently, we provide a sampling of the time axis at multiple levels of detail using uneven predefined intervals that match standard divisions of time: *day*, *workday/weekend*, *week*, *month*, *quarter*, *half year*, and *year*.

Once the time intervals are set, metric summaries at system level can be computed by aggregating system metric values on each sampling interval. This leads to another issue of software evolution visualization at system level: “How to aggregate system metric values in a meaningful way?”. For numerical metrics a number of alternatives are available:

- compute the average of all metric values. This approach provides an approximation of the overall evolution trend;
- select minimum/maximum metric value. This approach is useful for assessing evolution variability and identifying trend outliers;
- select first/last/ n^{th} metric value. This approach can be used to investigate evolution trends based on a given reference time stamp for each sampling interval.

For categorical metrics the aggregation can be performed by category based summarization. That is, each possible category gets assigned a value that represents the number of occurrences of category specific events or entities in the sampling interval. Many alternatives are possible for counting entities or events. We detail here only two of them, which facilitate the interface of system level evolution assessment with the file level approach described in Chapter 6. For each category entry one can count:

- the number of file revisions matching the entry. This approach is useful for correlating the category entry with the activity in the system;
- the number of files having revisions that match the entry. This approach is useful for correlating the category entry with the system size.

The total number of entities or events obtained by counting can then be divided with the total number of matches for any category entry to obtain normalized figures, for easy comparison.

For example, an *author* metric can indicate for each revision of a file who was the author that committed it to the repository. Assume one needs to sample the data obtained from an SCM repository to assess the system level evolution of the author metric. To this end one has to choose first a data sampling interval that matches the desired level of detail. Then, for each possible author, sampling is performed by counting the number of files containing revisions committed by the author in each interval. Eventually, for each sampling interval the obtained values can be divided by the total number of files, to obtain normalized figures. These can be used then to assess how the knowledge about the system is distributed on the team and how this distribution evolves. Such an assessment is useful for identifying critical developers in a project and for managing knowledge distribution risks. This analysis is demonstrated in Section 7.4.

7.3 Visualization Model

The main purpose of software evolution visualization at system level is to provide high level overviews of the system quality trends. Consequently, simplicity and clarity are key ingredients for building the visualization. The main target users for this type of visualization are:

- *managers*: users that are instrumental in organizing the software development and maintenance activities, yet lack the technical capabilities to base their decisions on low level information about the system contents. To this end, the correlation between trends of different quality indicators is essential for detecting anomalies and predicting future evolution;
- *evolution analysts*: users that investigate the evolution of a system in detail, for instance, using the techniques presented in Chapter 5 and 6. To this end, a high level overview on system evolution is useful to support the analysis process.

7.3.1 Layout and Mapping

We provide a visualization that follows closely the model proposed in Chapter 4. On the horizontal axis we encode time, and on the vertical axis we depict one entity, *i.e.*, the system (see Section 4.5).

Given the fact that the visualization depicts in this case only one entity, as opposed to hundreds of entities (see Chapter 5 and 6), there is enough space available on the vertical axis to encode more than one entity attribute. Consequently, two or more system quality indicators (*i.e.*, M^i metric sequences) can be visualized at each moment.

To this end, the straightforward solution is to use overlapping charts, such as line or bar charts (Figure 7.1a). This approach gives a compact overview of evolution, offers maximum resolution on the vertical axis for each attribute encoding, and enables trend based correlations between the visualized quality indicators. The main disadvantage of this solution is the difficulty of making absolute scale correlations between different quality indicators. On the one hand, using the same scale for all indicators might not be acceptable for visual investigation when value range differences are large across indicators. On the other hand, using different scales for each indicator on the vertical axis makes visual decoding difficult (*e.g.*, two points on the same vertical position but belonging to different charts have different values). To address these issues, a normalized scale can be used across indicators, but this does not support correlations based on absolute values.

To avoid the absolute value correlations problems of overlapping charts and to facilitate experimentation with different alternatives of encoding the evolution of one attribute, we chose to visualize quality indicators separately. Consequently, we integrated the resulting visualizations by stacking them on the vertical axis to enable time based correlations (Figure 7.1b).

Once a quality indicator is selected for evolution assessment, there are a number of alternatives available for visualizing it. Each of these alternatives can be used to perform

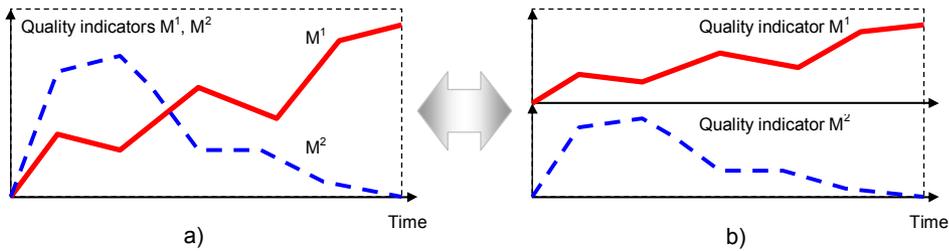


Figure 7.1: Visualization of software evolution at system level. Quality indicators are encoded in: (a) one image using overlapping charts; (b) separate images, stacked on the vertical axis

a specific analysis scenario. Below, we present some of them and we highlight possible usage scenarios.

- *bar chart*: the metric values are encoded on the vertical axis y using a constant interpolation between the sampled intervals (see Figure 7.2a). This type of encoding is suitable for giving a precise indication of what the value of a metric is at a certain moment.
- *graph*: the metric values are encoded on the vertical axis y using a linear interpolation between the sampled intervals (see Figure 7.2b). This type of encoding makes results more continuous and might allow seeing trends more easily when the sampling intervals on the horizontal axis are large.
- *intensity map*: the metric values are color encoded using a luminance map (see Figure 7.2c). This type of encoding could be used for detecting trends in a very noisy signal.
- *rainbow map*: the metric values are color encoded using a rainbow color map (see Figure 7.2d). This encoding could be used for detecting outliers when the space on the vertical direction is limited, and the resolution in the color map space becomes larger than the one in the screen space.
- *flow graph*: this is a special type of encoding intended mainly for summaries built on top of categorical metrics. It uses the same principles as the *ThemeRiver* metaphor proposed in [58] to show the relative occurrences of a number of categories in a set of files or versions committed in a specific time interval. To this end, color encoding (for categories) and space encoding (for numeric values) are combined in one visualization. In addition to the techniques described in [58] we use cushions to make the segregation between the flows more visible (see Figure 7.2e), even when the colors are very similar. User feedback from informal experiments indicates that this makes also the comparison between large and small quantities more comfortable.

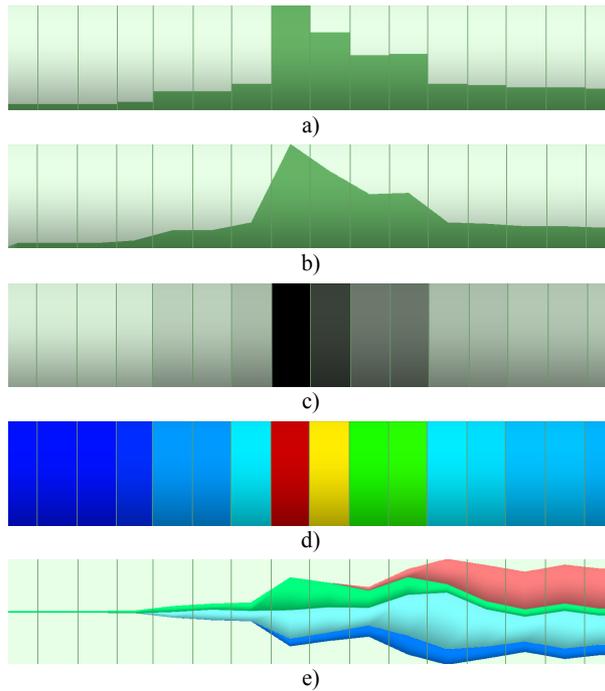


Figure 7.2: Alternative ways for encoding system metrics evolution: (a) bar chart; (b) graph; (c) intensity map; (d) rainbow map; (e) flow graph.

7.3.2 Visual Scalability

An important issue of the visualization proposed above is scalability. As presented in Section 7.2 the amount of data recorded by SCM repositories can be very large, and can cover many samples. By means of data sampling, users can reduce the level of detail at which they want to investigate the system evolution. However, it may happen that even after this step there is not enough space to depict the evolution on one screen. Additional measures are then necessary to obtain comprehensive overviews.

Outlier Enhancement along the Horizontal Axis

When a large evolution interval is covered, the number of sampling intervals may be higher than the number of available pixels on the horizontal axis. In such cases, the challenge is how to obtain a comprehensive overview of data of the horizontal axis. One possible way is to make use of antialiasing as presented in Section 5.3.3. Other alternatives have also been proposed (see [64, 85]). We wanted to explore new possibilities in this direction. Given the purpose of evolution visualization at system level, a method that quickly emphasizes outliers yet presents the evolution trend, is desirable. To this end, we use an additional dimension for encoding the presence and the characteristics of outliers on the horizontal axis.

In Section 7.3.1 we have presented a number of alternatives for encoding one attribute using either space or color encoding. Nevertheless, when depicting the evolution of an attribute using one dimension, we could use the other to emphasize the presence of outliers in the regions where more values need to be represented on the same pixel column. We use two alternatives:

- use vertical axis to encode attribute values and color to emphasize outliers;
- use color to encode attribute values and vertical axis to emphasize outliers.

In the first alternative, we use different colors to emphasize the presence of values above and below the average of a set of attribute values that share the same position on the horizontal axis (see Figure 7.3a). This straightforward alternative enables the user to observe the trend in the sampled set of values (via the average value), yet it highlights the presence of outliers, and the magnitude of the extreme cases. When the space available for showing differences between the average value and the extreme cases is too small to be observed, the outlier magnitude is considered to be irrelevant.

In the second alternative, we use three sections on the vertical axis to indicate with colors the maximum, average and minimum values of a set of attribute values that share the same position on the horizontal axis (see Figure 7.3b). This enable the user to follow the evolution trend in the middle section of the visualization on the vertical axis (via the average value), yet it highlights at the top and bottom the presence of outliers, and the magnitude of the extreme cases. When the difference in color between the three sections on the vertical axis becomes difficult to observe, the outlier magnitude is considered to be irrelevant.

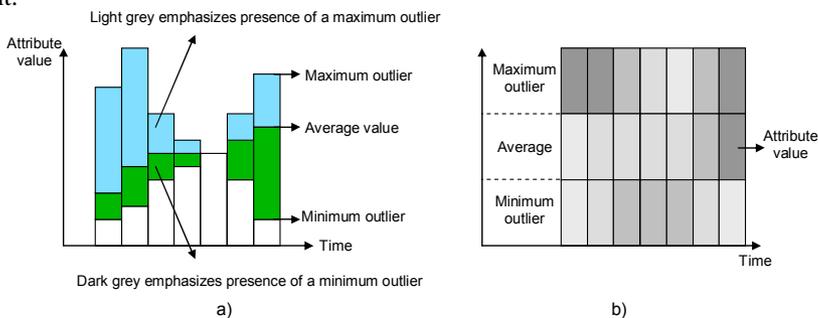


Figure 7.3: Outlier enhancement along the horizontal axis (principle): (a) attribute values are encoded using vertical axis, outliers are highlighted with colors; (b) attribute values are encoded using colors, outliers have specific positions on the vertical axis.

Both alternatives can be used to follow evolution trends, yet identify the presence and characteristics of outliers. The best alternative has to be chosen based on the concrete investigation scenario, using the guidelines presented in Section 7.3.1. However, the second alternative offers in general less resolution than the first one, and only relatively high magnitude outliers are easy to observe.

Figure 7.4 illustrates the use of the color encoded outlier enhancement technique with snapshots from a real-life project. The image depicts the evolution of the number of

files containing contributions from a specific user. This kind of representation can be useful when assessing how the knowledge about the system is distributed over the team members. The horizontal axis encodes time, the vertical axis encodes the number of files. Data is sampled at the day level. In Figure 7.4 top, the number of displayed days on the horizontal axis is much higher than the available screen resolution. Outliers are encoded using color. Light grey emphasizes the presence of maximum values, normal grey emphasizes minimum values. The area below the chart is filled with color (*i.e.*, dark grey) to make assessment of the overall trend easier. Three significant outliers in the thin vertical bars A, B and C are identified. The bars correspond to moments of high interference between the owners of the source code. The maximum outlier in bar A has the largest deviation from the average. In the corresponding period, the number of files has actually recorded also a decrease with respect to the previous period as the minimum outlier is less than the previous average value. Figure 7.4 bottom, shows a zoomed-in version of the image, focusing on the interval that contains the three bars. This shows the actual distribution of the values sampled by the highlighted bars A, B and C in the top image.

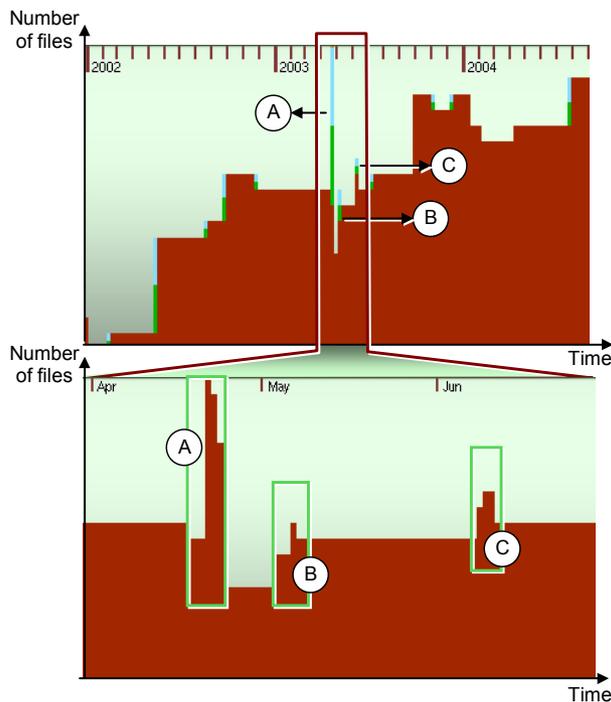


Figure 7.4: Evolution of the number of files modified by developer *X*: color outlier enhancement in bars A, B and C (top); zoomed-in version shows the actual value distribution in A, B and C (bottom).

Range Scaling on the Vertical Axis

Metric value ranges may be very large. If precise value estimations are important for assessing the evolution of a metric, space should be chosen to encode metric values on the vertical axis (*i.e.*, using bar charts or graphs). Nevertheless, the available space on this axis is also limited. One problem arises: “How to increase the vertical resolution when detailed analysis is required on a small interval of the total value range?”. When the total available space is limited, there are several alternatives available for addressing this issue.

One alternative is to increase the space available for encoding the interval of interest at the expense of decreasing the space available for the remaining part of the total range. In the extreme case, the space available for the remaining part can be discarded entirely. This approach, however, has the disadvantage of hiding context information, which might be important during analysis. Another approach is using deformations of the vertical axis, *e.g.*, a 1D fisheye lens (see Figure 7.5b). This approach provides both a higher resolution for the interval of interest, and an indication of the value position in the remaining range. The drawback in this case is that different scales are present in the same image, which makes it difficult to compare values variations in the detailed region with those from the context area.

Another alternative to increase the vertical resolution is to share space between intervals. In this approach the total range is divided in a number of intervals that share the vertical axis, and values are displayed using interval dependent colors (see Figure 7.5c). For every value, the intervals that correspond to smaller ranges are drawn first as vertical stripes with interval specific color, covering the entire available space in the vertical direction. This creates the impression of the chart representation being “wrapped” around the available vertical space, with every layer having a different color. A similar visual appearance can be achieved following the two-tone color map technique described by Saito *et al.* [94]. The space sharing approach cannot be combined with the outlier enhancement technique previously described, as both make use of color. Additionally, the visualization of highly discontinuous metrics is difficult when the number of co-located intervals is larger than two. However, when the data is relatively continuous, this approach offers both an overview of the overall value trend, and detailed information on the entire value range.

Figure 7.6 illustrates the use of range scaling on the vertical axis with snapshots from a real-life project. Similar to the example presented in Figure 7.4, the image depicts the evolution of the number of files modified by some developer. In the top image, the limited resolution makes it difficult to assess the evolution in the highlighted area. The bottom image uses range scaling with shared intervals to depict the same information. The original range of 120 files is split in two intervals, 1-30 and respectively 31-120, such that maximum resolution on the vertical axis is achieved for the highlighted area. Light grey is used to encode the first interval and dark grey to encode the second. The second interval is drawn on top of the first one. The evolution of the values in the range of the highlighted area can be assessed by looking at the first interval. This is easier than in the first case as more screen space is available for the visual representation. Additionally, the second interval helps in constructing the overall context, showing the evolution of the values that are higher than those in the highlighted area.

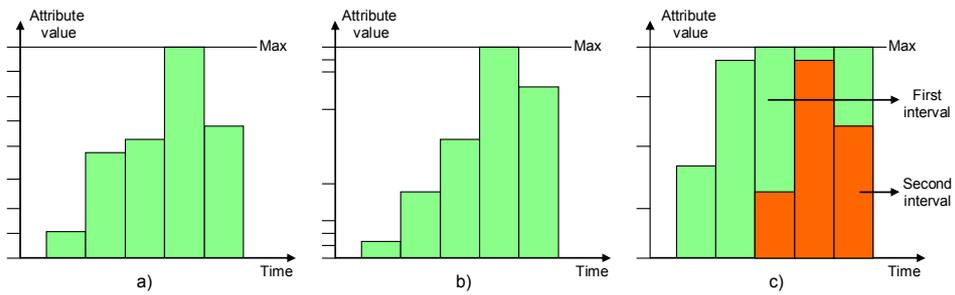


Figure 7.5: Range scaling on the vertical axis (principle): (a) no scaling – difficult to assess the differences between records 2,3 and 5 ; (b) scaling using a deformation (1D fisheye lens) of the vertical axis; (c) scaling with vertical axis sharing – color encodes the interval.

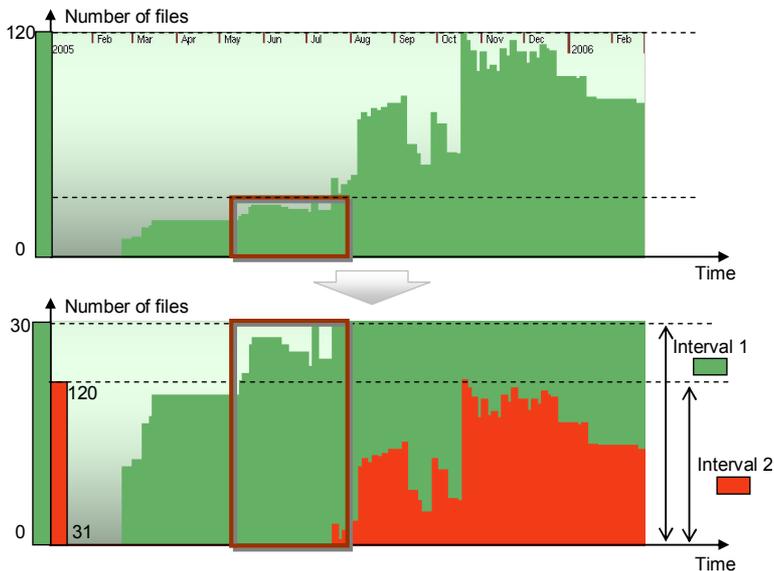


Figure 7.6: Evolution of the number of files modified by developer X: no range scaling on the vertical axis makes it difficult to assess evolution in the highlighted area (top); range scaling with two shared intervals increases resolution for detailed investigations in the highlighted area, yet conserves the context information (bottom).

7.3.3 User Interaction

Interaction is an important component of software evolution visualization at system level. Two main aspects have to be considered in this respect: how to indicate what is to be shown and in which way, and how to make correlations with other visualizations.

In the previous sections we have detailed a number of alternatives that can be used to filter the data and to build graphical representations of it. All presented alternatives have both advantages and drawbacks. The suitability of a specific approach for addressing a

given issue depends very much on both the problem at hand and the user. Therefore, it is important to enable users to easily switch between alternatives and find the most appropriate one. To this end, we propose an interaction mechanism that keeps users' focus on the visualization. We use the mouse wheel and control buttons on the keyboard to enable the user to rapidly switch between the preset sampling intervals, without moving the mouse cursor. Additionally, we use a pop-up menu to provide the user with the list of possible visual encodings. Consequently, the user does not have to move his focus from the visualization in order to choose the one that suits best his needs. The only operation that requires moving the user focus point is range scaling on the vertical axis. A slider is provided for this, attached to the visualization. While this approach forces the user to move his attention point, the focus still remains within the boundaries of the visualization (see Figure 7.7).

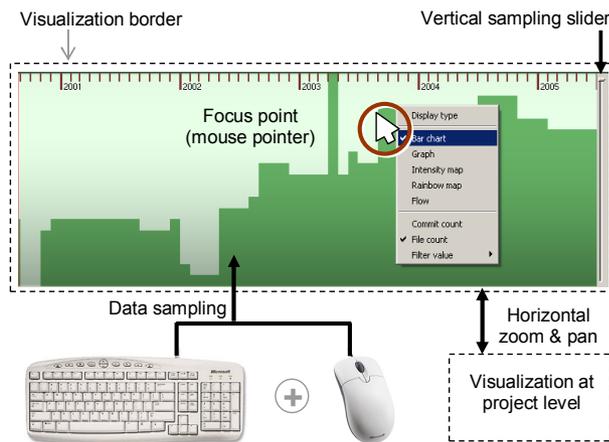


Figure 7.7: Interaction mechanisms for software evolution visualization at system level. The focus point is maintained within the boundaries of the visualization. Zoom and pan operations are correlated with other visualizations.

Another important interaction issue is the synchronization of the visualization at system level with other visualizations of software evolution. The visualization we propose in this chapter can be used as starting point for detailed analysis on the evolution of software. To this end, one can use the visualizations depicting software evolution at line and file level presented in Chapter 5 and 6. A correlated views environment depicting evolution at multiple levels of detail is, therefore, preferable. In such a case however, maintaining consistency across visualizations has to be enforced. To address this issue, we propose using a synchronized zoom and pan mechanism on the time axis for all visualizations depicting the same software entity, independent of the level of detail. Consequently, both the software visualization at file level and that at system level will have the same zoom and pan coordinates when they address the same set of files. While this maintains consistency between views, it also facilitates the process of focusing on a specific evolution interval, which has to be done only once (*i.e.*, for one view).

In the next section we illustrate the applicability of the techniques described above on real-life data with a number of practical experiments. These experiments try to answer the

questions formulated in Section 7.1 using different visualizations of software evolution at system level.

7.4 Use-Cases and Validation

To evaluate the use of the visualization approach proposed in the previous sections, we implemented it as an add-on application to the CVSgrab tool presented in Chapter 6. Then we used the resulting application in a number of experiments to assess the evolution of real-life projects at system level. Next we detail the visualizations we constructed and the insight we gained from three of these experiments.

MagnaView

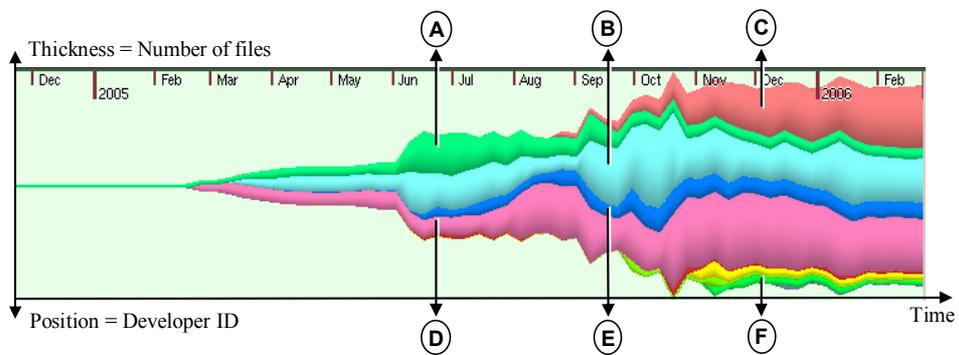


Figure 7.8: Assessment of distribution of the software knowledge over the development team in MagnaView.

The first project we investigated is MagnaView [74], a commercial visualization software package containing 312 files, written by 11 developers in over 16 development months. We visualized the evolution of this package at system level to assess the distribution of knowledge in the development team. First we sampled data at week level. Then we used a flow graph to compare the number of files each developer owned (*i.e.*, files last committed by the developer). This can give an indication about what users are familiar with the up-to-date contents of the source code in a project. Figure 7.8 presents an annotated version of the image we obtained. One can see that in the first three months the code was owned by only one developer (A) and was relatively very small. At the end of February, 2005, three more developers joined the team: B, D, and E. Developers A, B and D seem to own equal amounts of code in the following months, while developer E has little influence. At the beginning of June 2005, the code size increases notably, and developer E becomes owner of a small part of the code. The amount of code owned by E does not change significantly until the end of the observed period. At the end of August 2005, a new developer joins the team: C. Until the end of the evolution period, the amount of code owned by this developer increases steadily. In the same time the amount owned by developer A, the starter of the project, diminishes significantly. At the end, the code is mostly owned by three developers: B, C and D. One could assume these three developers have a good

understanding about the system and are important members of the team. The amount of code they own is also equilibrated, so the risk of losing one of the key developers is reduced to a minimum. The amount of code owned by A and E is very small. However, each of these two developers owns as much code as the remaining 6 developers together (F).

mCRL2

The second project we investigated is mCRL2 [81], a tool set used to specify and analyze the behavior of distributed systems and protocols. This software package contains 2635 files, implemented by 15 developers during more than 28 months. Out of these, 371 files are C and C++ code files. We visualized the evolution of these source code files at system level to assess how easy to maintain the project is. First we sampled the data at week level. Then we used a number of visual mappings to investigate the evolution of code size (*i.e.*, number of lines of code) and the evolution of cyclomatic complexity (see [79]). Figure 7.9 presents an annotated version of the images we obtained.

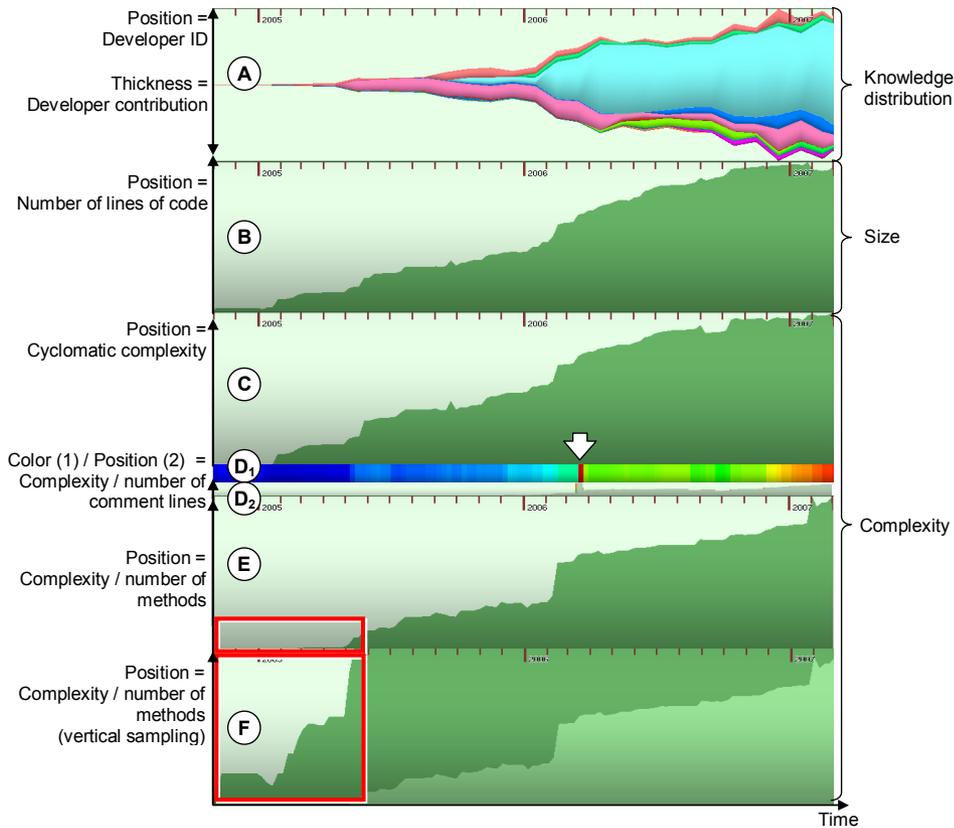


Figure 7.9: Assessment of project maintainability in mCRL2: A - knowledge distribution; B - code size; C - total code complexity; D_1/D_2 - total complexity divided by the number of comments available in the source code; E - total complexity divided by the number of methods in the source code; F - same as E but uses range scaling with shared intervals.

Image A depicts the distribution of knowledge in the project, similarly to the previous example. One can see that after the second half of the first development year, an important developer joins the team. Until the end of the investigated period, this developer becomes the owner of most implemented code. Consequently, he is the only one up-to-date with additions and changes in most part of the project. From this point of view he represents an important asset for the project, but also a high risk. If this developer leaves the team, a very high amount of knowledge has to be transferred to other team members.

Images B and C show graphs that depict the evolution of the code size, and respectively the evolution of the total cyclomatic complexity of the system. One can notice that the two software metrics are highly synchronized. Therefore, one can be used to assess the overall trend in the other. Additionally, one can see that the code size and its complexity appear to have stabilized in the last five months of the observed period. This may be an indication that efforts are focused now on corrective maintenance, and a new release of the tool set might be soon available.

Images D_1 and D_2 depict the evolution of the ratio between the total complexity and the number of comment lines available in the source code. Image D_1 uses a rainbow color mapping. Blue encodes a low ratio, red encodes a high ratio. Image D_2 uses a bar chart. One can notice that the ratio becomes higher as the time passes, as more effort is spent on implementing new functionality and less on documenting existing code. From this point of view, the system becomes harder to maintain. The surprising fact is that the ratio continues to grow even at the end of the observed period, when the size and the complexity appear to have stabilized. This suggests that existing comments have been removed in this period. Consequently, these comments might have represented dead functionality that was cleaned up during the stabilization phase. Another surprising aspect is the outlier highlighted in the image. By correlating it with the graphs depicted in images B and C, one could infer that new functionality of high complexity was first added to the system, and then documented. Indeed, by inspecting in detail the commit logs stored on the mCRL2 repository, one can discover that the outlier corresponds to a tentative addition of the `LTS` library to the system, which was consequently fully integrated and documented.

Both image D_1 and D_2 use a very narrow vertical space. We showed both images to a number of persons in an application demo setting. The informal feedback we received from several attendants was that the rainbow color map was quicker in conveying the overall evolution trend and the presence of the outlier. However, when more space is available on the vertical direction, the bar chart encoding appears to become more efficient and conveys more accurately the same information.

Eventually, images E and F show graphs that depict the evolution of the ratio between the total complexity and the number of methods in the system. Image E gives a classical overview. One can observe that in the middle of the development period and towards the end, the ratio increased significantly in a very short interval. This signals either the addition of new functionality to the existing interfaces, or the addition of new high complexity code. Indeed, a detailed analysis of the commit log reveals that the jump of the ratio in the middle of the development period was caused by the addition to the repository of a high complexity library for C++: `boost`. Additionally, the jump at the end was caused by a high volume of changes and the addition of the `tcpip` library. However, by inspecting

image E one cannot easily assess the evolution of the ratio in the highlighted area. It seems as if the ratio would not change even if the complexity and the code size change (see images B and C). This and the very low values of the ratio suggest that the entire activity was dedicated to interface declaration in the highlighted period. Image F gives better insight in this matter. By using a range scaling on the vertical axis with shared intervals, the space available for inspecting the evolution of the ratio in the lower range is significantly increased. Consequently, one can notice the change in code size in the beginning of year 2005 (see image B) causes a three times increase in the investigated ratio. This suggests, that not only interface declarations took place in this period but also actual functionality was implemented.

KDE KOffice

The third project we investigated is the KOffice application suite of the Open Source project KDE [66]. We next present the outcome of our assessment using the same type of visualizations as for the previous project. Our aim is to use the software evolution visualization at system level as a way to compare quality aspects of the two projects.

KOffice is a collection of office productivity tools including among others a text processor, a spreadsheet and a presentation making utility. The entire software package contains 10616 files, implemented by 270 developers, during more than 9 years. We focused our analysis on an important part of this system: the `libs` folder. This contains a set of common libraries used across multiple applications of the KOffice suite, such as the user interface library `kofficeui`. The libraries are implemented in 843 C and C++ code files. We visualized the evolution of these source code files at system level to assess how easy to maintain the `libs` folder is. First we sampled the data at month level. Then we used a number of visual mappings to investigate the evolution of code size (*i.e.*, number of lines of code) and the evolution of cyclomatic complexity (see [79]). Figure 7.10 presents an annotated version of the images we obtained.

Similarly to the previous two experiments, image A depicts the distribution of knowledge in the project. Assessing this image one can observe that significant project owners appear from the second year of development. In the third year, four developers have the role of major code owner, and the amount of code they share is equally distributed. In the next four years however, one of these developers becomes more active, and takes over the code owned by others. From this point of view, he becomes a critical asset for the project. Two major discontinuities can be identified in the image at the end of the observed period. They signal project-wide code reorganizations that are not related to code owning, but are meant to ensure consistency in style. These type of reorganizations are typically performed by one developer, whose influence decreases rapidly after committing the code. After the second discontinuity however, one can observe that the influence of the previous major code owner is significantly reduced, while the code is distributed among five other developers. This may be an indication that the previous owner left the project and his knowledge was distributed to a number of remaining developers. Additionally, the amount of code owned by key developers increases with respect to the previous years. This is an indication of a more active code management policy in the project, centralizing knowledge from a larger number of developers. Consequently, the project should become easier to maintain.

Images B and C show graphs that depict the evolution of code size and cyclomatic

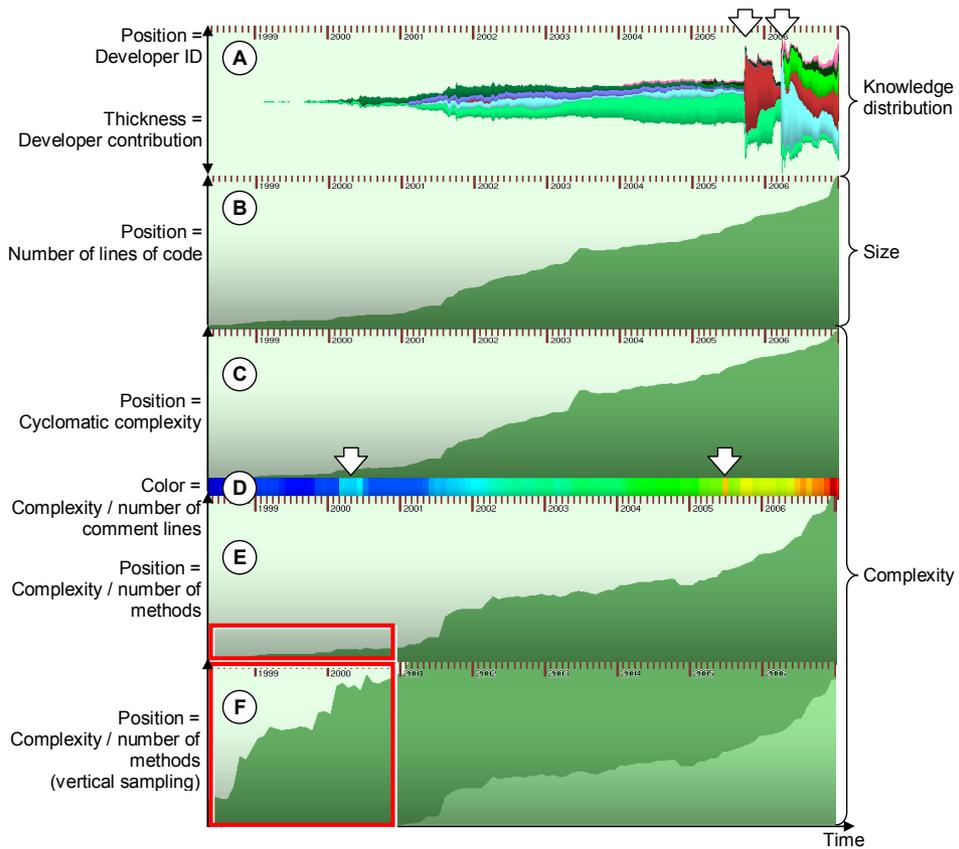


Figure 7.10: Assessment of project maintainability in KDE Koffice: A - knowledge distribution; B - code size; C - total code complexity; D - total complexity divided by the number of comments available in the source code; E - total complexity divided by the number of methods in the source code; F - same as E but uses range scaling with shared intervals.

complexity. Similarly to the previous example, these two metrics are highly correlated. Consequently, one can be used to assess the overall trend in the other. However, compared to the previous example, both the code size and complexity appear to increase significantly at the end of the investigated period. This may be an indication that the development efforts are focused on implementing new functionality and not on corrective maintenance. Consequently, a new release of the software package might not be available soon, and a lot of effort will be required next to take the project through a stabilization phase.

Image D uses a rainbow color mapping to depict the evolution of the ratio between the total complexity and the number of comment lines available in the source code. The color encoding scheme is the same as the one used in the previous case. Similarly to the second project, the ratio becomes higher as the development progresses. Consequently, more effort is spent on implementing new functionality and less on documenting existing code. This makes the system harder to understand, and therefore, harder to maintain.

The ratio grows significantly at the end of the investigated period, in the same time with the previously observed increase in code size and complexity (see images B and C). This suggests that the next stabilization phase of the project will require a lot of development resources. Two outliers are also indicated in the image. The first outlier corresponds to a high amount of additions to the source code. The second outlier corresponds to importing reusable functionality in the libraries from other parts of the system. Both outliers correspond to complex code additions that are subsequently documented.

Images E and F show graphs that depict the evolution of the ratio between the total complexity and the number of methods in the system. Image E gives a classical overview. One can observe that in the middle of the fourth development year the ratio increased significantly after a short stagnation period. This suggests that the development focused on adding new functionality to the existing interfaces. The increase is correlated with the more active code management policy observed in image A. Consequently, one may infer that the project strategy recorded a significant change in the fourth development year. However, by inspecting image E one cannot easily gain insight in the period preceding this change (*i.e.*, the region highlighted in the image). It seems that the ratio had a steady increase, which is characteristic to mature projects with a regulated development process. Image F gives better insight in this matter. By using a range scaling on the vertical axis with shared intervals, the space available for inspecting the evolution of the ratio in the highlighted area is significantly increased. Consequently, one can notice that the ratio recorded two significant jumps, when a lot of functionality has been implemented. Additionally, the ratio had also a number of decreasing periods, when implemented functionality was removed from the system. These patterns can be identified also in the previous project. They are characteristic to loosely planned and managed projects, like many of the Open Source projects are, especially in the beginning of the development period.

7.5 Conclusions

In this chapter we have presented a set of methods and techniques that can be used to build visualizations of software evolution at system level. Visualizations of this type are intended for quick assessments of evolution to identify overall trends or to trigger more detailed investigations.

Software metrics are the main source of data to be visualized. These metrics are commonly accepted as quality indicators for software systems. We addressed the challenges of building a visualization for the evolution of these indicators at different stages of the visualization pipeline. First we analyzed the type of software metrics that can be visualized, and we provided a preset based mechanism for sampling data. Next, we presented a set of layout and mapping alternatives that can be used to visually encode data, and we gave a set of guidelines for using them. We proposed encoding only one metric per visualization, and using the time axis to correlate evolution trends across multiple visualizations. Subsequently, we addressed visual sampling related issues both on the horizontal and vertical axes, such that data is presented to users in a meaningful way. Eventually, we proposed an interaction mechanism that tries to keep the focus point of users within the boundaries of the visualization, for making more efficient use of the short term memory.

To validate the utility of the proposed visualization, we used it in an number of experiments to assess the evolution of real-life software projects. During the investigations we focused on maintainability related issues. The visualization we constructed revealed a number of interesting aspects:

- Many projects have a number of code owners, *i.e.*, developers that are responsible for the code in a large number of files. For mature projects, the amount of knowledge should be equally distributed over a number of code owners to minimize the risk of losing it when these assets leave the team;
- Code size and code complexity are highly correlated in most projects. Therefore each of these metrics can be used to estimate the evolution trend of the other;
- In all investigated projects the ratio between code complexity and number of comments in the source code increases. This suggests increasingly more effort is spent on implementing and less on documenting code. As a consequence, projects become more difficult to understand and they require more effort for maintenance.

For the visualization proposed in this chapter we have made a number of assumptions about the nature of the data. Sequences of software metrics are assumed to be rather continuous in time. This enables the identification of trends and outliers, using the techniques described in Sections 7.2 and 7.3. In principle the same techniques can be applied to other sequences that are non time-related, for instance, the sequence of results of a software metric applied on all source code files in a project. Depending on the specific metric, the sequence might not be continuous on the file axis. Further investigation are required to assess the suitability of the presented techniques for this type of sequences.

As another direction for future research, formal user studies need to be organized to get a deeper understanding about the performance and suitability of the presented data encoding, outlier enhancement and range scaling alternatives for specific investigation scenarios.

The visualization of software evolution at system level is intended for high level investigations in the trends of software metrics. They cannot reveal detailed aspects about the evolution of the source code. For this, the visualizations at line and file level presented in Chapters 5 and 6 can be used. However, a correlated view environment that links all visualizations in one application would make the evolution assessment more efficient. System level visualizations could be used to trigger detailed investigations within the same environment following the visual information seeking mantra: “overview first, zoom and filter, then details on demand” [96]. In this respect, the visualization of software evolution at system level can be used as a generalized version of the horizontal metric views presented in Chapters 5 and 6.

Chapter 8

Visualizing Data Exchange in Peer-to-Peer Networks

In this chapter, we verify the applicability of the visualization techniques and methods proposed in Chapters 5, 6, and 7, across the border of the software evolution domain. To this end, we present a novel visualization for the performance assessment of peer-to-peer file-sharing networks. First we identify the relevant data transferred in this kind of networks and the main performance assessment questions. Next, we describe the visualization of data from two different points of view: the server view and the file view. Based on shaded cushions, we introduce a novel technique: faded cushioning. This technique allows visualizing the same data from different perspectives. To correlate the server and the file views, we provide a special scatter plot. Finally, we discuss the effectiveness of the presented visualization and the applicability to this case of the techniques presented in the previous chapters.

8.1 Introduction

In the previous chapters we have presented the evolution of software structure at three levels of detail, commonly used by the Software Engineering community and readily available from SCM systems. In this chapter we use a similar approach to address dynamic aspects related to software execution. Our goal is to investigate the suitability of the techniques proposed in the previous chapters, across the borders of the software evolution domain. To this end we focus on software for a particular type of distributed systems: Peer-to-Peer (P2P) file-sharing networks.

Distributed systems consist of a number of network connected nodes that cooperate for solving a given task. Their distributed nature, however, makes them difficult to understand. In this respect, visualization can facilitate getting insight. Most work in this area is related to the visualization of structure of distributed systems [9, 36]. Systems' performance is, however, one of the less explored issues. In this chapter, we present a novel

approach to the visualization of performance of P2P file-sharing networks. This type of networks is a branch of distributed systems that has recently gained enormous popularity. The presented visualization techniques are illustrated by EZEL, a prototype tool that we developed for the assessment of performance in the ED2K P2P file-sharing network [35]. A copy of the tool can be downloaded from [42].

In Section 8.2 we present the issues that are relevant for the assessment of performance in P2P file-sharing networks, with a focus on the ED2K. In Section 8.3, we describe the data that is transferred in this kind of systems, and we identify the transactions that are important for performance evaluation. Next, we detail the challenges that arise when supporting the assessment with visual tools, and we present a novel approach to address them. In Section 8.4.1, we propose a visualization of data taking servers as focal points. In this section we show how, via the use of shading and color, multiple aspects can be presented simultaneously in a compact way. Elaborating on the space partitioning power of cushions, we introduce a novel technique: fading cushions. We demonstrate how this technique allows visualizing the same data from different perspectives. In Section 8.4.2, we add the viewpoint of the file, and in Section 8.4.3 we show the correlation between file and servers via a special scatter plot. Finally, in Section 8.5 and Section 8.6 we discuss the suitability of the presented approach for the assessment of P2P file-sharing networks, and we reflect over the similarities with the approaches we presented in Chapter 5, 6, and 7.

8.2 Problem Description

A P2P file-sharing network is a collection of computers organized in an ad-hoc network with the purpose of sharing digital content. To this end, connected computers rely primarily on the storage capacity, computing power and bandwidth of the other participants in the network rather than on a relatively low number of central servers. Next, we outline the most important concepts in such a system with an eye on their implementation in ED2K. Figure 8.1 shows a conceptual model.

Clients generate requests, *e.g.*, file read requests, and assign them to proxy entities. A *proxy* divides requests in smaller parts, *i.e.*, segments, that are uniquely identifiable and can be independently fulfilled by *server* entities. Every proxy has an internal dispatcher algorithm that decides to what servers the requested segments will be sent for processing. Every server has limited processing resources to handle request segments from proxies, and uses a priority based scheduling to manage them. The priorities are internally maintained by the server for each client request.

Visualization of a distributed system's performance aims at helping the user to understand such a system, based on information obtained from transactions between its constituent parts. Both snapshots and history recordings are therefore important [77]. The user can employ this understanding to navigate the transaction data and answer a number of performance related questions. In the case of a distributed file-sharing systems, one is mainly interested in two issues: dispatcher algorithm and server performance.

Dispatcher algorithm assessment

When the network of processing servers is large and dynamic, *e.g.*, P2P networks, the segment dispatching algorithm has a strong influence on the request servicing time. The

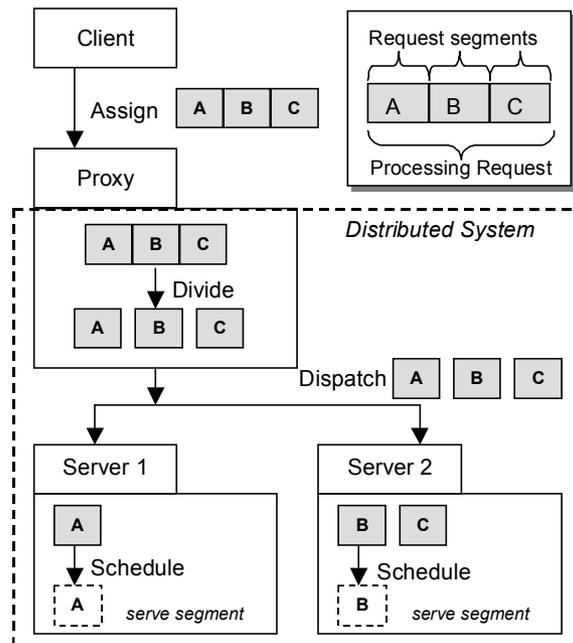


Figure 8.1: Distributed processing system (conceptual model)

performance visualization should help users to easily assess the dispatcher algorithm, and reveal the factors and the circumstances that might influence it. For example, users should be able to identify the reasons for which a slower server is selected at a certain moment instead of a faster one.

Server assessment

When the dispatcher algorithm on the proxy allows direct selection of the servers, performance visualization should help to determine which server delivers the best value. The interesting case appears when the selection is based on a number of independent performance figures. The most important questions and quantities relevant to P2P networks are:

- *download speed*: how long does it take till one gets a requested file?
- *server popularity*: how long do clients wait in the server-side queue, and how frequently do other clients with higher priority enter that queue?
- *server specialization*: what kind of requests can a server satisfy?

When assessing the performance of a P2P file-sharing network, one has to investigate the evolution of a number of independent parameters. An effective assessment should consider the loosely coupled parameters together, and should be based on tradeoffs that depend on the purpose of the assessment. The very nature of tradeoff making requires the user to divide his/her focus over more assessment criteria at once. This turns out to be rather difficult when the number of criteria becomes higher than two. A typical download

session for a 700 MB movie file contains around 200,000 transactions. If one uses just standard time graphs to visualize the above three quantities, the overall image is quickly lost, and the dispatcher algorithm and server assessment questions remain unanswered. The challenge is to build a unified visualization, in which the user can focus on a particular quantity of interest without losing overview.

For P2P file sharing networks, four main criteria to assess a server can be used:

- download speed (higher is better)
- size of segments (larger is better)
- queue evolution (fast advance and less re-queuing after admittance is better)
- segment position (depending on the download purpose, some segments may be more important than others)

The ideal server should be fast, able to provide large contiguous segments, and should have a small waiting time. Additionally, it should not be very popular, to reduce the chance that other clients with a higher priority interrupt the download by acquiring the server. However, such servers usually do not exist. Moreover, the assessment depends on several characteristics of the downloaded file, as explained next. For a fast download of a small file, such as a 3 MB MP3 music file, selecting the fastest server may not be the most appropriate decision. When the waiting time in the queue of the fast server exceeds the time that another slower server requires to perform the task (*i.e.*, including the time for waiting in the queue), the slower server is preferred. Another example is the download of an archive, for instance, a ZIP file. Such a download should not be attempted from a server providing fragmented segments, even if it is fast. A slower server that provides contiguous segments is preferred, as it makes archive recovery simpler when the download cannot be completed.

In the following sections, we present the challenges of building a visualization tool for P2P file-sharing networks. The proposed solutions to these challenges are illustrated with snapshots from EZEL, a validation visualization tool that we developed for the performance assessment of the popular ED2K P2P network.

8.3 Data Model

The first issue one has to consider when building a visualization tool is which data to visualize (see Chapter 4). P2P file-sharing networks are characterized by a large number of terminals connected via the Internet. Each terminal connected to such a network can act both as a server and as a client at the same time. Clients generate file read requests that proxies break down into segment requests. A segment request is fulfilled by a single server, which provides the client with the related file segment. A file segment consists of file blocks and has a variable size (expressed in blocks).

All terminals in the network exchange transactions based on a specific protocol. These transactions may contain either file blocks, or control information (*e.g.*, download requests, file availability info, queue evolution info). In the case of the ED2K network,

the exact protocol in use is not disclosed, which makes the assessment task considerably more difficult.

As mentioned in the previous section, server and dispatcher algorithm assessment are central issues for performance evaluation of P2P file-sharing networks. We address these issues by analyzing the transaction data that a client exchanges with the rest of the network.

To study the dynamic behavior of servers, we record two types of transactions: *file block arrivals* and *queue position reports*. With this information, we build three functional descriptions for a server, from the point of view of a given client. In the following, it is assumed that a client is serviced by NS servers S_1, \dots, S_{NS} , every server S_i being identified by an integer server id. The download time t runs from 0 to the download completion moment T_C . The three server descriptions are:

$$\text{Queue position: } Q(S_i, t) : \mathbb{N} \times \mathbb{R} \rightarrow \mathbb{N}$$

Gives the position of the client segment request in the queue of server S_i at time t . If $Q(S_i, t)$ is zero, the client can start downloading from S_i .

$$\text{Download speed: } V(S_i, t) : \mathbb{N} \times \mathbb{R} \rightarrow \mathbb{R}$$

Gives the speed with which the client receives data from the server S_i at time t .

$$\text{Contribution: } C(S_i, t) : \mathbb{N} \times \mathbb{R} \rightarrow \mathbb{N}$$

Gives the data downloaded from a server S_i from the beginning till a given time t . In other words:

$$C(S_i, t) = \int_0^t V(S_i, \tau) d\tau$$

The total amount of downloaded data is thus:

$$D = \sum_{i=1}^{NS} C(S_i, T_C)$$

To assess the performance of the dispatcher algorithm, one has to consider both the server assessment and the evolution of the downloaded file itself. For that, we record the *block arrival* events and we correlate them with the *file segment requests*. With this information, we construct three functional descriptions of a download:

$$\text{Provider: } P(p) : \mathbb{N} \rightarrow \mathbb{N}$$

Gives the server that provided the block at a position p , for all positions p in a downloaded file.

$$\text{Time of arrival: } T(p) : \mathbb{N} \rightarrow \mathbb{R}$$

Gives the moment when the client received the block at position p , for all positions p in a downloaded file.

$$\text{Segment: } S(p) : \mathbb{N} \rightarrow \mathbb{N}$$

Gives the file segment to which the block at position p belongs to, for all positions p in a downloaded file.

The quantities mentioned above are discrete. For example, a typical movie download consists of around 200,000 time moments t , $NS=150$ servers, and a total downloaded value of $D=700$ MB.

All above functional descriptions are equally important for the performance evaluation of a P2P file-sharing network. Consequently, the challenge is to build a visualization that facilitates access to all of them and shows how they relate to each other. In the next section we propose such a visualization.

8.4 Visualization Model

The goal of P2P data exchange visualizations is to assess the dynamic behavior of individual servers, view how a file is downloaded, and see the relation between these processes. Since the functional descriptions to be visualized have several implicit non-trivial dependencies, a straightforward visualization (for instance using separate graphs) is not a good solution.

Given that the set of functional descriptions has three main axes (Servers S_i , Time t , block Position p), a visual representation using a 3D scatter plot may appear to be a direct solution. Figure 8.2 depicts such an approach. Every dot represents the transmission of a block from a server at a certain moment. However, this visualization would be very hard to interpret, given the large amount of time samples (hundreds of thousands), the inherent 3D occlusion problems, and the data scattering.

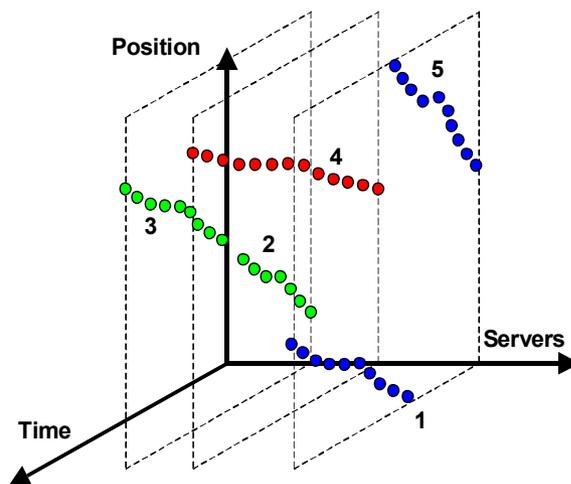


Figure 8.2: 3D visualization for P2P performance assessment

Therefore, we split the visualization in two parts correlated via a scatter plot: one focusing on servers, the other on the downloaded file. The server visualization is de-

scribed next. The downloaded file visualization is described in Section 8.4.2. Finally, Section 8.4.3 presents the custom made scatter plot.

8.4.1 Server Visualization

To support the assessment of servers with a visual representation, we use a horizontal sequence of small diagrams, one per server. This allows the user to easily compare the functional descriptions of different servers (*i.e.*, Q , V and C). Additionally, the representation of each server should offer enough provisions to relate it to the visualization of the downloaded file (Section 8.4.2). There are several alternatives for an individual server representation. The obvious choice is to use the horizontal axis for Time, the vertical axis for *Queue* (Q) and *Contribution* (C) and to display their variation as graphs (Figure 8.3a). The *Download speed* (V) can be estimated in this setup from the slope of C .

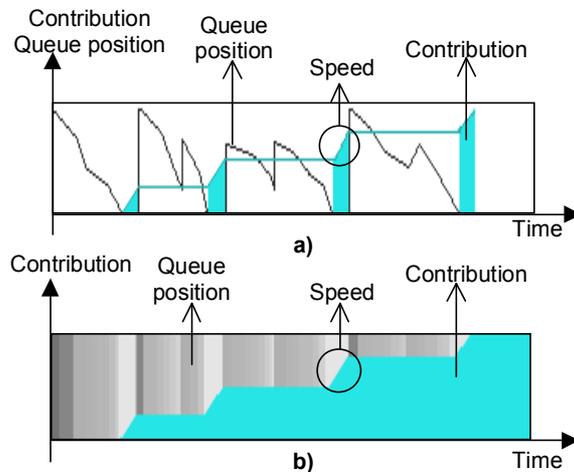


Figure 8.3: Server diagram: (a) with graphs only; (b) with graph and luminance strips.

However this first alternative is quite noisy for real world cases. Due to the mutual exclusion in time of downloading and queuing, the evolution of *Queue position* and *Contribution* are not continuous, but interleaved. To remove the noise from the visualization, we replace the spatial encoding of *Queue position* with a luminance encoding. Rectangular strips whose gray shade indicates queue position are used (darker shades indicate lower positions). Although graphs are more precise, grayscale encoding of the queue position is sufficient for the stated purposes. After all, the user needs only to identify the overall position and to spot general queue trends such as advance or high / low position alternations.

Additionally, we use solid color filling for the area under the C graph to enhance the feeling of quantity that *Contribution* has. Figure 8.3b depicts the result of the second approach. Both C and Q variations appear now continuous, which makes interpretation easier. Moreover, while their representations do not interfere, they still allow users to easily make correlations. The horizontal parts in the variation of C , for example, indicate

periods in which the *Contribution* stagnated. The user can easily verify if queuing was the cause of idleness, and can also check the queue evolution of the segment request in that period. Similarly to the first approach, the *Download speed* evolution can be estimated from the slope of C .

The next visualization design step is to arrange the server images such that they allow easy comparative assessment. For this, a way to easily distinguish and identify the diagrams is necessary. To this end, we use color encoding, and in each image, we fill the area below the *Contribution* graph with a server dependent color. Color allows one to easily distinguish the different diagrams and also preserves server identity over changes in the diagram arrangement.

To allow easy comparison of the server diagrams, they have to be arranged (sorted) along one of the spatially encoded axes, *i.e.*, the *Time* axis or the *Contribution* axis. Using the *Time* axis for arranging the server images (Figure 8.4) proves to have two major drawbacks. First, it is hard to compare server quantities (queue position, contribution) at the same given time instant. For example, one could hardly decide if the contribution of a source exceeds that of another, at a given time t_0 (Figure 8.4).

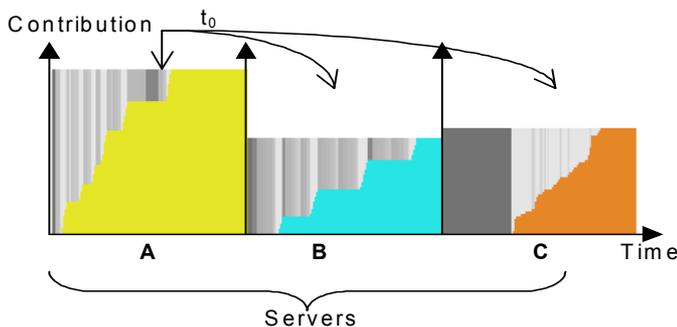


Figure 8.4: Server diagram arrangement along the *Time* axis

Secondly, the time interval (width of server diagrams in Figure 8.4) is identical for all servers, so no meaningful comparison could be made along the *Time* axis itself.

The second alternative (*i.e.*, arrange on *Contribution* axis) is better, as it allows easy comparison of servers based on their total contribution (Figure 8.5).

Additionally, for a given time t , this allows comparing the queue position and the cumulated contribution to that moment t .

Figure 8.5 presents a typical visualization obtained with the method presented so far: a file download served using five servers. It is easy to see that the first server (purple) is the most productive one: it gives about 50% of the total amount (half of the horizontal axis), has a stable throughput (constant slope), client requests are promptly getting on the first queue position, and maintain this position for the total download duration (purple image slope has no step-like jumps, and its queue area has a constant light shade after we get on the first position). It is also easy to identify in this image the less productive servers, *i.e.*, the slow one (orange) and those exhibiting frequent falls in the queue position (yellow and cyan).

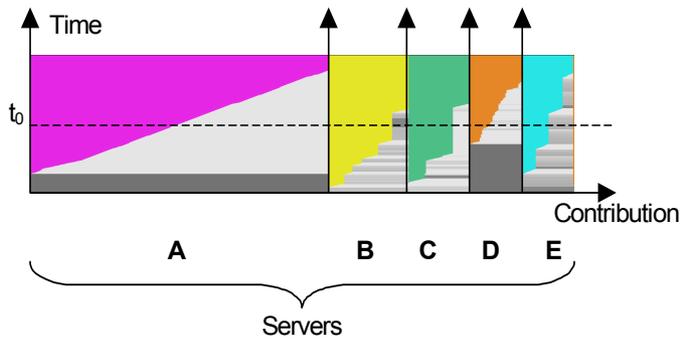


Figure 8.5: Arranging server diagrams along the *Contribution* axis

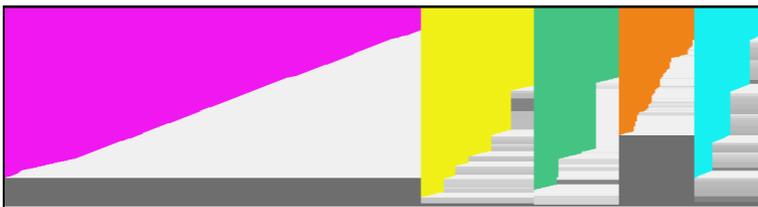


Figure 8.6: Basic server visualization

However, this visualization is still limited. First, using only color to encode server identity is not a good solution when the server arrangement (horizontal axis sorting) can change. It may happen that two servers with the same, or perceptually similar, colors are arranged one next to the other (Figure 8.7 top). Indeed, it would be preferable to use only a few (10..16) perceptually different colors, whereas typical network configurations have over 150 servers. Using only luminance (gray value) to encode the queue position causes similar problems. On the other hand, color encoding of server identity keeps visual coherence when rearrangement occurs.

Spatial Partition with Bi-level Cushions

We solve the above mentioned problem using the space partitioning properties of cushions. For a detailed description of cushions, see [111]. As depicted in Figure 8.7, cushioning makes separation clear between different servers encoded with the same or similar colors, without using extra screen space. It also delineates the borders where the difference in luminance makes distinction hard.

In the above server diagrams, the total contribution of a server consists of a set of segments. As the size of the segments varies, it would be preferable to visualize it. That would be also useful later on for making correlations with the download visualization.

With the server visualization presented so far, it is hard to figure out the individual segments, as they are encoded using the same color (*i.e.*, the color of the server). To emphasize the segment partitioning inside the diagram of a server, while maintaining

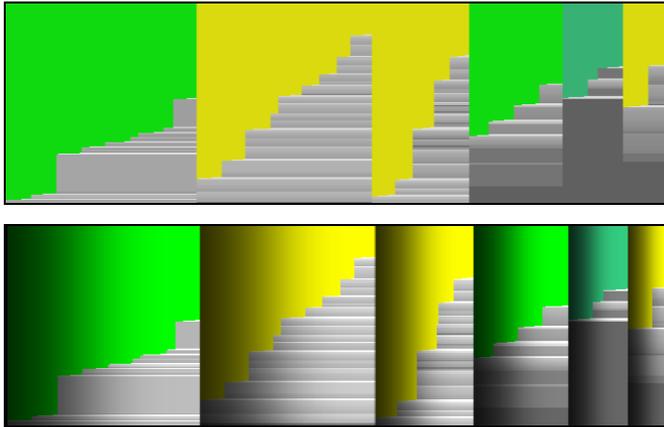


Figure 8.7: Server arrangement : without cushioning (top); with one level cushioning (down).

clear separation between servers, we use the bi-level cushioning technique described by Van Wijk and Van de Wetering in [111]. Figure 8.8a depicts the main idea behind this approach. By each server diagram the illumination of a height-modulated surface is visualized. The height assigned to a point in a server diagram is the sum of two parabolas (*i.e.*, cushions), one that describes the server, and one that describes the segment to which the point belongs. The surface is illuminated using a spot light that forms an incidence angle with the normal on the base plane. Each server diagram depicts the image projected by light reflection on a plane parallel with the base.

Figure 8.8b depicts the result of this technique. By using OpenGL texturing, a much higher performance is obtained than the similar software-only implementation of Van Wijk and Van de Wetering [111]. In detail, the server rectangle image and each of its segment rectangle images, as in Figure 8.7 top, are blended with a 1D texture containing the respective server or segment luminance profile in the alpha channel.

Focus Migration with Faded Cushions

Using the bi-level cushioning is very effective for delimiting servers and segments within servers. However, the above method draws cushioned segment information also over the area that displays queue information (gray area in Figure 8.9). Segment partitioning is not relevant for that area, and this makes server comparison based on queue evolution, *i.e.*, following horizontal correlations, difficult.

In order to maintain the desired segment and source partitioning effect, and, at the same time, remove the undesired influence on the queue evolution visualization, we extend the bi-level cushioning. We change the perceived shape of the segment cushions in the vertical direction from constant curvature to a gradually flattening profile. To achieve this, we introduce a height variation in the vertical direction using a decreasing profile as sketched in Figure 8.10a. For this profile, we use an asymptotic function (*e.g.*, $y^{\frac{1}{n}}$). The

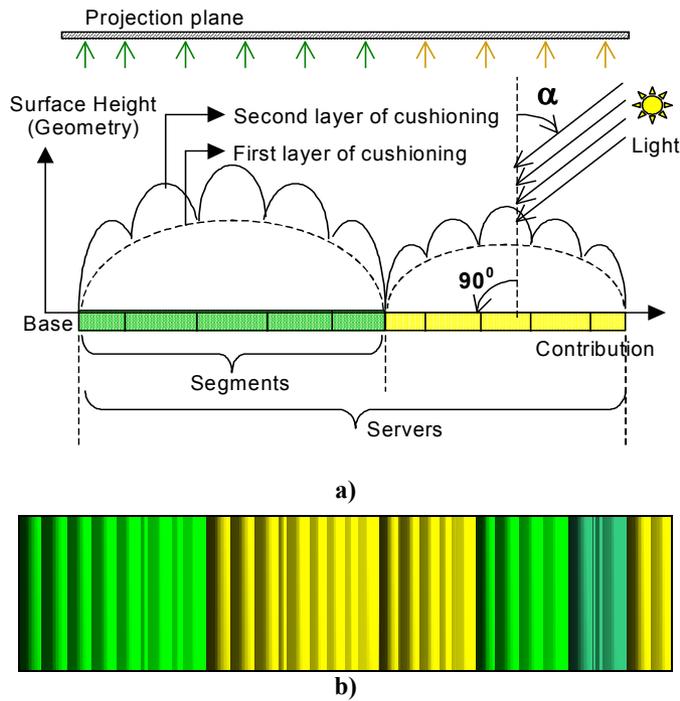


Figure 8.8: Bi-level cushioning for segment and source partitioning: (a) principle; (b) result.

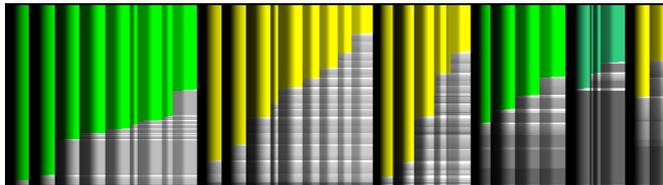


Figure 8.9: Basic bi-level cushion visualization

segment cushions are efficiently implemented as 2D alpha textures and blended atop of the original 1D server cushions.

Eventually, we obtain a visualization that emphasizes both segment and server segregation at the top of the image, and then progressively focuses only on the partition in servers, as the user's focus moves to the bottom of the image. The gradual transition makes focus migration smooth while preserving the server context (Figure 8.10b). In other words, the visualization exhibits vertical coherence at the top (segment-server area), which smoothly changes into horizontal coherence at the bottom (queue area). The overall visual effect resembles the draping of a curtain, and nicely scales up for visualizations containing up to 200 segments on a common computer screen (*i.e.*, with a resolution of 1024×768 pixel).

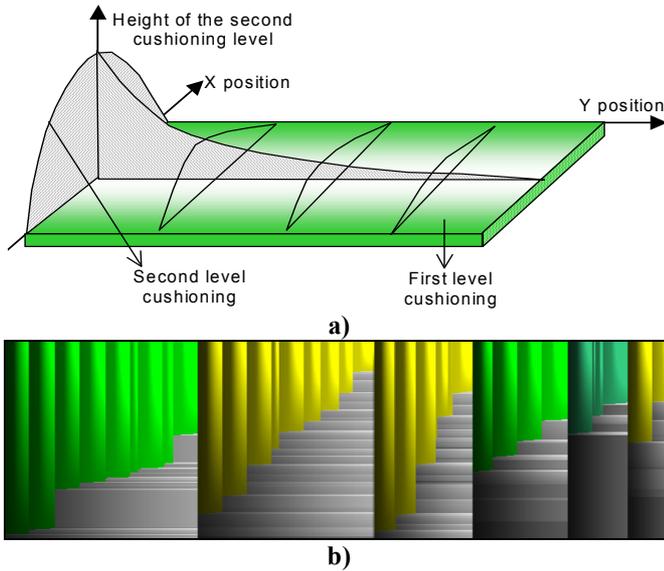


Figure 8.10: Enhanced bi-level cushioning for smooth focus migration: (a) principle; (b) results.

8.4.2 Download Visualization

In this section we address the visualization of the download itself and the creation of correlations with the server visualization described in Section 8.4.1.

The only alternative in this part is to use the block *Position* as one of the main axes in the representation, and report the functional descriptions to it. The challenges are, however, in choosing the right visual encoding for the *Provider* (P), *Time of arrival* (T) and *Segment* (S) descriptions. To make correlation with the server visualization easy, we use color to encode P , and we choose the same color assignment as for the server visualization.

For *Segment* encoding (*i.e.*, S), we use a similar approach with the one from the server visualization: one-level cushions are built on top of fixed-width rectangles arranged along the *Position* axis (Figure 8.11). Bi-level cushions are not necessary, as the emphasis is only on segment segregation, and has to be visible along the entire width of the rectangles.

For *Time* encoding, a graph-like representation may be considered. Neighboring segments on the *Position* axis, however, may arrive at non-adjacent time intervals which immediately leads to a very noisy visualization. Therefore, we use a rainbow colormap ($t = 0$ is blue, $t = T$ is red) to encode the time on a per segment basis (Figure 8.11). While this alternative is visually less accurate for identifying the arrival time of a block, it consumes little space and attenuates the visual noise caused by neighboring segments that arrive at different moments in time. Moreover, the above color scheme highlights discontinuities, *i.e.*, segments that arrive at moments distant in time with respect to their neighbors. To improve the image generation speed, the time is not shown for every single

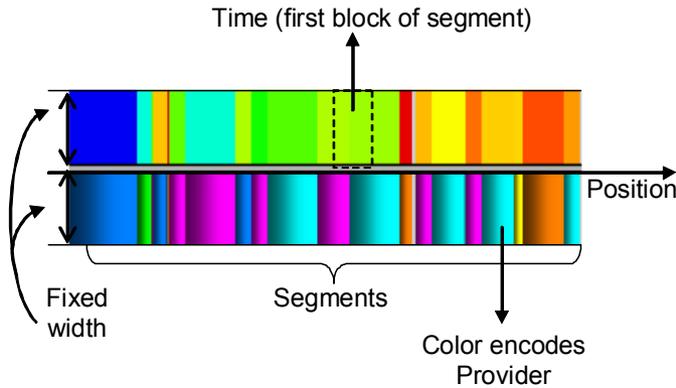


Figure 8.11: Visual encoding of functional description for a file download

block in a segment, as the T description specifies. Instead, for all blocks in a segment we use the same time description as for the first block, and a more accurate representation is implemented through server correlations, which are describe next.

8.4.3 Correlation Visualization

In this section we present how to visualize correlations between the server and download visualizations. In the design of the visualization so far, a color-based correlation exists already between the *Provider* description (*i.e.*, P) and the server diagrams. This allows to identify and compare servers that provide some particular blocks in a downloaded file.

Next to this, a correlation that would make the T description more accurate is also necessary. Since the server visualization has a good mapping from *Time* to *Contribution* (*i.e.*, C), we extend this mapping to the download visualization through a correlation along the block axes (*i.e.*, the *Contribution* and the *Position* axes). However, given that the two axes are spatially encoded, a relation at block level would be too fine-grained and hard to visualize. For that reason, we visualize the connections at the (higher abstraction) segment level.

The discrete nature of the block axes favors using a scatter plot representation to visualize the correlation. A simple scatter plot, however, makes visual associations difficult, once the number of segments is greater than 10 (Figure 8.12a). A possible workaround is to add lines that make connections explicit. However, this alternative proves to be ineffective too, as it clutters the image, and suffers from aliasing once the distance between lines becomes too small (*e.g.*, the black line in Figure 8.12b). These problems are only aggravated by the large number of correlations (hundreds) that must be displayed for a standard download dataset.

In order to make the connections more explicit while keeping the image uncluttered, we replace the solid lines with shades that start from the points of the scatter plot and fade away as they approach the axes (Figure 8.12c). This alternative reduces the confusion created by crossing lines, and offers still enough visual clues for recognizing connections.

Additionally, it introduces no artifacts and scales very well with the image size. When the distance between the points of the scatter plot becomes too small to observe differences, the shades merge naturally, as if they were addressing the same element. To accomplish this, the shades are drawn using `GL_MIN` blending function of OpenGL, which always keeps the darkest shade element at intersections, regardless of the drawing order.

To maximize the efficiency of screen real estate usage, we choose a metric view format for the download visualization. This trades the size requirements of the download visualization on the horizontal axis for a higher segment resolution in the server visualization.

The complete visualization, obtained after linking the server and download visualizations using the correlation methods described in this section, is depicted in Figure 8.13. For easy navigation, interactive selection facilities are added to allow restricting the download visualization part and the corresponding correlations to:

- specific parts of a file (by individual segment selection on *Position* axis)
- specific time intervals (using a time cursor on the *Time* axis)
- specific servers (by individual server selection on *Contribution* axis)

These selection mechanisms easily allow one to answer questions such as:

- *Which are the servers active at a given time moment?*
- *Which are the file blocks provided by a given server?*
- *Which are the servers a given file part came from?*

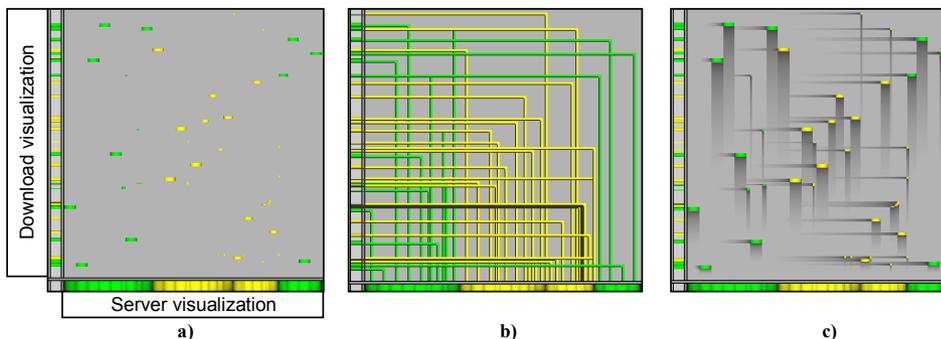


Figure 8.12: Correlation visualization alternatives (a) basic scatter plot; (b) adding connecting lines; (c) adding shading.

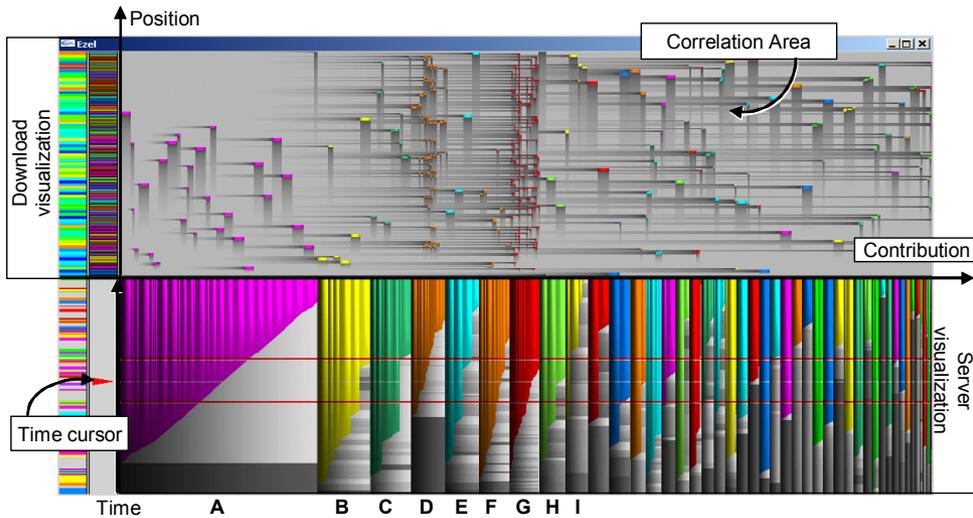


Figure 8.13: EZEL: A visual tool for the assessment of performance in P2P file-sharing networks

8.5 Use-Cases and Validation

In order to validate the visualization techniques we propose, we used EZEL to assess the performance of the ED2K network with a number of concrete cases. We present here two of these investigations.

In order to experiment with the tool, we needed real-life information about transactions in ED2K. We obtained such datasets by instrumenting eMule [39], an open source download client for the ED2K network. The instrumented client provides a log file from which the functions Q , P , C , V , T , and S discussed in Section 8.3 may be computed.

Figure 8.13 shows a visualization of the download of a large movie file (702.4MB). The complete download took several hours and contained 201,261 transactions. In this image, the servers are sorted in the decreasing order of their total contribution. The upper half of the image shows the segment fragmentation on a per server basis. According to this visualization, the most suitable download sources for archive files are A, B, E and H, as they provide large sets of contiguous segments, which makes archive recovery simpler in case of incomplete download. The least preferred in this sense are sources D, F and G, which provide tiny segments scattered along the entire length of the file. Analyzing the slopes in the image (*i.e.*, the server speed) one can see that I is one of the fastest sources. Unfortunately, it is also a very popular one, as most of the time the client request waited in the server queue. A better alternative, especially for the download of a small file, is using servers B, C, F or G. Although F and G are slow and provide fragmented segments, they are unpopular, and thus start satisfying the client requests very fast. Finally, if one were asked to single out an overall good download source, A would qualify, as it provides many data, with constant throughput, and little waiting time.

Figure 8.14 depicts a situation where a weakness of the dispatcher algorithm was spotted. For a downloaded file (350MB) the servers were arranged in decreasing speed order.

In Figure 8.14 bottom, the display of segment evolution in time was switched off. Using a time cursor, segments that were downloaded at a certain moment t_0 close to the end of the download were selected. Figure 8.14 top shows that at t_0 the downloaded segment came

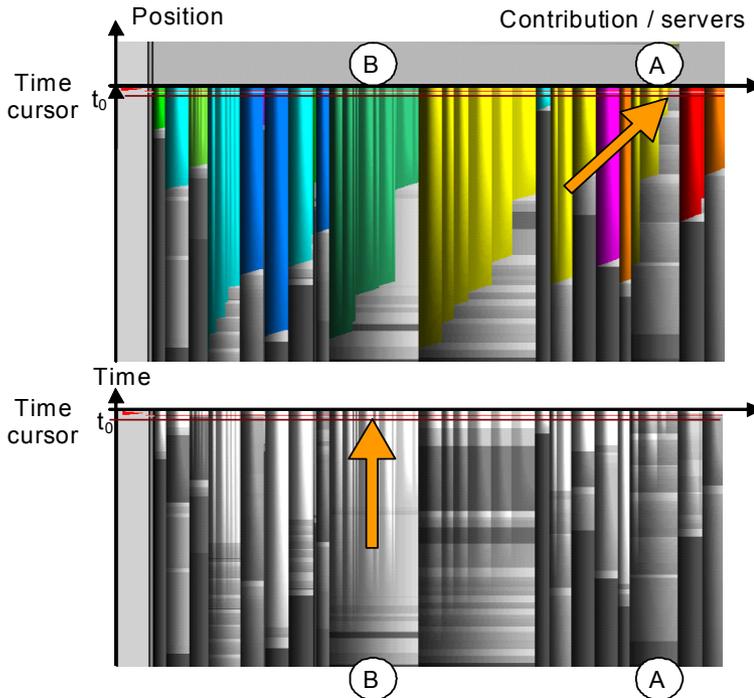


Figure 8.14: Dispatcher algorithm assessment. At time t_0 source A provides data (top) although the faster source B is also available (bottom).

from server A (*i.e.*, the contribution slope of A indicates data transfer), while Figure 8.14 bottom shows that at the same time, the faster source B was also available as the client request was not in queue but ready to be served (*i.e.*, the corresponding position in the queue of B is white). That means the dispatching algorithm in the eMule client is not optimized for the minimization of waiting time.

The examples we presented in this chapter address the visualization of data obtained at the end of a download. However, it is possible to use the same approach for building a dynamic, real-time visualization of the acquired data. The only issue in this case would be the frame rate at which images are produced. In the worst case, more than 200,000 transactions have to be considered for the generation of each frame. This would require an impressive processing power in order to update the image at the arrival of each new transaction (on average every 100 ms). The current implementation of EZEL generates images at 0.5 - 1.0 fps on a Pentium 4 processor running at 2.6 GHz. A scenario in-between would be to generate images on demand with the partial information available during the download. While differences between consecutively generated images could be too large to make meaningful correlations, the individual images may be used to gain insight in the download process and interact with it based on intermediate assessments.

8.6 Conclusions

In this chapter we have presented a novel approach for the visual assessment of performance of P2P file-sharing networks, with a focus on the ED2K network. First we have identified the data transferred in such a network, and then we have constructed relevant performance descriptions based on it. Subsequently, we have built a custom visualization made of two correlated parts: a server and a download visualization. For each part, we offer a number of visually encodings based and enhancements and combinations of existing visualization techniques. Notably, we use shaded cushions for virtually all data elements (servers, segments, queue positions, and correlation plot elements). Overall, this novel visualization gives a compact and scalable way to present a download consisting of thousands of transactions, from over 100 sources, on a single screen. We validated this approach using EZEL, a prototype assessment tool.

To the best of our knowledge, this is the first attempt to visually explore the data transfer dynamics in the rapidly growing world of P2P file-sharing networks. The other work in this field addresses a different task, namely visualizing the topology of a P2P network of a different type [127]. In the future, this visualization could be generalized and extended to the larger domain of distributed processing in general. The challenges we foresee there are related to the process visualization of the dispatching and scheduling entities.

The visualization we have proposed in this chapter does not address the evolution of software, but a different domain, *i.e.*, data exchange in P2P networks. However, while building this visualization we used some of the (or similar) visual techniques with the ones proposed in the previous chapters.

The shaded cushions technique we used in Chapter 5 through Chapter 7 to segregate file versions, files or subsystems, proved to be very efficient in segregating file segments and sources in the visualization proposed in this chapter. This confirms the segregating power of cushions and its generality across application domains. However, the faded cushions technique that we introduce in this chapter is, in retrospect, not applicable in the visualization models proposed for software evolution. There, the second layer of cushioning, which segregates files inside clusters, has to be visible along the entire file length. While this makes cluster segregation a little bit more difficult to visualize, it enables file segregation at all times, which is not necessary when dealing with P2P data exchange.

Another reoccurring aspect when building visualizations of software evolution and of data exchange in P2P networks is the use of color. In all visualizations we built, color is used to encode identity. While in software evolution visualizations, color can be used to encode the identity of the artifact creators, in P2P networks color encodes the identity of the segment providers. In both cases color offers an intuitive way of correlating identity with structure and time. Both cases suffer also from the inherent limitations of this approach. While users can efficiently perceive up to 12-16 individual colors, there are potentially much more identities to be visualized at a specific moment. To cope with this drawback, a given color map is used only as a first level of visual pattern detection. When interesting patterns are detected, the color map can be modified to ensure a unique encoding of the entities that generate the pattern. Consequently, interesting patterns can

be first identified and then validated with this two step approach.

Metric views are also one common aspect across software evolution and P2P data exchange visualizations. The limited space graphics summarizing information along a structure or time axis enable powerful correlations along that axis, without making domain specific assumptions.

The layout of time evolving entities is also similar for software evolution and P2P data exchange visualizations. Both the contribution of sources and the evolution of lines, files, and systems are arranged along a time axis, in an orthogonal layout for easy comparison. In both cases the horizontal axis is used to encode the dimension that requires better resolution according to the application usage scenarios. While in software evolution the time axis is horizontal, in P2P data exchange visualizations the time axis is vertical. However, in both cases the order of entities along the second (perpendicular) axis can be interactively adjusted using sort operations. This enables easier comparison of entities, assessment of aggregate metric distributions, and outlier detection for specific metrics.

One aspect where the software evolution and P2P data exchange visualizations differ is the visual encoding of multiple attributes. While in the case of software evolution, multiple attributes of an entity are encoded using alternating color maps and composable textures, the P2P data exchange visualization uses fixed color maps. This is an application specific aspect, due to the nature of the encoded attributes and the use-cases they are involved in. The server diagrams for example depict the evolution of both server contribution and client request position in the queue. They might be encoded using different color mappings. However, in this case, the two entities are highly dependent: contribution changes only when the client request is on the first position in the queue. Therefore, one can assume there is no change in the queue position while contribution changes, so the two quantities can receive an alternating spatial layout. Other correlations are implemented using metric views and the novel scatter plot. These support the identity based investigations, which form the main use case-base for the proposed P2P data exchange visualization.

In the next chapter, we address the reoccurring aspects of building software evolution visualizations in more detail. We also look at these aspects from the point of view of their applicability in other domains, and we try to give a set of recommendation for their broader applicability.

Chapter 9

Lessons Learned

In this chapter we give an overview of the recurrent issues that we identified when building visualizations of software evolution, and we present a set of guidelines and solutions for addressing them. We start with the data acquisition and preprocessing problems. Next we present a number of design aspects that we identified as useful when designing visual representation of software evolution. These aspects may have a broader applicability in designing visualizations of other abstract, complex evolving data types besides software. Finally, we discuss the problems we encountered when trying to evaluate our visualization approach.

9.1 Data Acquisition and Preprocessing

We implemented the visualization model we propose in this thesis in a number of tools for assessment of software evolution. As data source for these applications, we used history recordings we retrieved from SCM repositories. Such repositories are actually the only available instruments at the moment that record the intermediate states a software system has during its life cycle. Additionally, SCMs are widely accepted by the software engineering community, and are considered to be key ingredients for successfully managing large projects [16]. This makes them very suitable for empirical research on software evolution.

However, using data from SCM repositories for analysis is not straightforward. The main goal of SCM repositories is to enable collaborative work in a distributed project and to keep back-up copies of software. To this end, changes are recorded almost always at a very low level, *e.g.* text line for CVS, byte for Subversion, and not all evolution patterns are explicitly stored. Therefore, in order to assess the evolution of software based on information stored in SCMs, one has to recover the evolution patterns of interest at the desired level of detail from the available information (see the *integration* challenge in Section 1.3).

Some patterns are explicitly stored on the repository, for instance the file insertion,

continuation and deletion in Subversion. Some other patterns can be built on top of the stored information. For example, line insertion, continuation and deletion can be constructed based on the textual difference operator built in CVS and Subversion. There are, however, evolution patterns that are not recorded by or easy to extract from the two SCM systems that we investigated. Merge patterns, for example, are completely overlooked, in spite of the fact that many SCM systems have specific tool support for conducting merging activities. These patterns have to be detected by mining the existing information. This is a difficult task acknowledged by the research community that needs further investigation before it can be used in practice (see [45, 48, 52, 82]).

Another problem of assessing software evolution based on the information stored on SCM repositories is the fact that SCMs support a single representation of software. However, software can have many orthogonal representations, for instance as a hierarchy of files, a set of interrelated classes or an abstract syntax tree. For assessing the evolution from the perspective of another representation, one has to recover first the evolution patterns for the representation provided by an SCM (*e.g.*, software as a set of files) and then to translate them to the desired one (*e.g.*, software as a set of interrelated classes). This translation cannot always be done in a manner that achieves the same results as if the evolution patterns were recorded specifically for the desired representation. The main reason for this is the semantic loss that takes place when converting between different representations of software, in combination with a semantic interpretation of evolution. For instance, questions such as “will a class continue to deliver the same functionality after some of the lines of code implementing it changed?” are very difficult to answer. Translation heuristics can be applied to improve the results, yet their efficiency depends on the specific application scenario.

A possible solution to all problems mentioned above could be the creation of a new type of SCM repository that specifically records a wider range of evolution patterns for a wider range of software representations. The recording should be done at the moment when evolution takes place and should accommodate semantic interpretations of evolution.

9.2 Software Evolution Visualization

The visualization model we propose in this thesis has to address an instance of one of the most difficult problems acknowledged by the information visualization community: the visualization of large multivariate data sets.

Useful visual assessment scenarios of software evolution require at least the detection of structure evolution patterns. These can then be correlated with a plethora of software attributes to gain insight in the evolution of the system. In this respect, a simple mental model of the data and an efficient usage of the short term memory are a must. Last but not least, the entire visualization application must be user friendly and intuitive for its typical users (see the *usability* and *intuitiveness* challenges in Section 1.3). To address these challenges, we identified a number of design aspects in our visualizations of software evolution that appeared to have a positive contribution during the informal user studies and experiments we organized. We present these next.

A 2D representation of data has several advantages with respect to a possible 3D alternative. First, it is occlusion free. Secondly, 2D representations combined with zoom and pan operations are easily understood by software engineers accustomed to work with 2D software design spaces (*e.g.*, UML diagram editors).

A simple 2D layout, using time on the horizontal axis and software entities such as lines or files on the vertical one is easy to learn and scales well with large amounts of data. Interactively constructed layouts via selection, sort and cluster operations on the available data attributes enable the user to guide the creation of a mental map of software evolution. Additionally, they might reduce the clutter in the image and facilitate the detection of outliers when they match a natural ordering of the visualized data (*e.g.* ordering files in the evolution view based on the file creation time in Chapter 6).

Multiple views using the same type of 2D orthogonal layouts, arranged to share one axis, are effective in finding correlations via the shared axis. To this end, metric views can be used. These views summarize the information presented in a main view using a limited screen space, and enable users to discover high level correlations.

Carefully crafted textures can be used together with color to encode several attributes at the same time at the same screen location. To be able to identify textures, however, a minimal display area is required for the encoded surface. From our experience we found an area of 20×20 pixels to be the minimum value in this respect. Several textures (*i.e.*, up to three) can be overlaid to show even more attributes. For obtaining the best results, the texture patterns should be designed such that they are orthogonal to each other, and also they should not be aligned with the axes of the entity layout.

Colors are very efficient for encoding categorical attributes, either ordinal or nominal. They are particularly helpful when building and adjusting interactive layouts, as they facilitate the update of the mental model by preserving category identity. From our experience, good results can be obtained for nominal attributes when using color to encode up to 15 different categories. The category uniqueness, however, cannot be perceived when more than 15 categories have to be color encoded. With ordinal attributes, good results are achieved when using a rainbow map to encode up to 10 different value ranges.

Geometric shaded cushions have very good visual partitioning properties. They can strongly emphasize the segregation of visualized entities while their interference with the color encoding is less distractive than by using border lines. We have noticed that the exact shape choice of the cushion profile to use is largely a matter of concrete application and, not in the last instance, of personal user taste.

A preset controller based on the mechanism proposed by Van Wijk and Van Overveld[112] can enable correlations between color encoded attributes. One can use the controller to switch between alternative color encodings, using a distance based weighted blending. By controlling the distance between the elements of the controller, the user has full control over the blending process. The user can continuously paddle between the elements, which leads to the formation of a mental map of the possible color encodings, and enables finding correlations.

A user interaction design that preserves the position of the attention point avoids the pollution of the short term memory and helps maintaining the mental map of software evolution. Pop-up menus and combined mouse and keyboard actions are key ingredients

for such designs. Several informal user experiments showed that the use of the mouse wheel in combination with control buttons from the keyboard is perceived by the user as a natural way for implementing zoom operations. Preset zoom levels are also highly appreciated, and they are best placed in the area immediately bordering the visualization. The inner mouse button (*i.e.*, left for right handed people) appears to be best used for performing selection operations (in combination with the keyboard). Pop-up menus are good for giving layout adjustment commands.

Color legends and labels of layout axes are instrumental for maintaining a mental model in a multiview environment, and for using visualization as a communication means. Legends for explaining the color encoding appeared to be the most important ones in our informal user experiments. They were followed by labels for indicating scale and for sampling axes. Our users perceived these as 'must have' annotations in all visualizations we designed.

All above guidelines should be considered together when building visualization of software evolution. However they do not make specific assumptions on the data domain. Consequently, they might be helpful when building visualizations of other types of multivariate abstract data as well. In this respect we used the same design elements, *i.e.*, 2D representation of data with orthogonal interactive layouts, metric views, cushions, identity encoding color schemes and a focus preserving interaction mechanism to assess the data exchange mechanisms in a Peer-to-Peer network (see chapter 8). The same designed elements were used by different researchers in building a visualization application for dynamic memory management analysis (see [83, 84]).

Another important challenge of software evolution visualization is the huge amount of data to be visualized. Presenting the entire information on one screen is impossible. Methods and techniques are required that support data and visual sampling, yet allow detailed investigations of the evolution data (see the *scalability* challenge in Section 1.3). In all visualization we propose in this thesis we identified three design aspects that can help creating comprehensive overviews. We discuss these next.

First, a correlated multi-view environment presenting evolution information at multiple levels of detail allows focus + context investigations without the need for continuously adjusting visualization parameters. The correlation between views is best implemented by using the mouse to indicate in the context view the area that needs to be investigated in more detail, and consequently by visualizing that area in the focus view.

Secondly, deformations of the horizontal and vertical axes allow a redistribution of the available screen space, and increase the resolution when assessing a specific part of the visualization, while maintaining the context information. Non-constant deformations (*e.g.*, fish eye lens) can be misleading sometimes. While they increase the resolution in particular areas, they make it very difficult to decode spatially encoded properties due to the non-uniform scaling (*i.e.*, distance proportions change during deformation). This method is good, nevertheless, for detecting outliers. Constant deformations, for instance a simple lens, and shared axis interval representations are more suitable for investigations requiring distance estimations. Some specific deformations of axes can be, nevertheless, acceptable even for space distribution assessments. In these cases, however, a different data model is assumed via deformation. For example, in Chapter 5 we used a custom deformation to switch between a time and a version uniform sampling of the time axis.

While the time uniform sampling addresses evolution from the perspective of a number of calendar related events, the version uniform sampling assumes data is a sequence of evolution steps.

Thirdly, screen space antialiasing preserves visualization structure across multiple levels of visual detail and is very useful to eliminate sampling artifacts when visualizing a large amount of data on a limited screen space. This lets us draw colors which express several data attribute values per single pixel. However, this type of antialiasing does not work for visualizations using a rich set of different hues. The blending of different colors may lead to the creation of already assigned tones with non-related meaning and that might lead to wrong insights. The same observation was made for a similar application in a related domain by Moreta and Telea [84]. A possible solution to this problem is using a carefully designed color map which allows meaningful color interpolation and averaging. Another alternative is to perform importance based antialiasing, *i.e.*, select a single data sample from all that are to be drawn on the extent of one pixel, and draw only that sample using the associated color encoding. Clearly, the very definition of this method suggests it is highly application and scenario dependant, as there are many ways of defining what “importance” is. However, this does not preserve the structure of the visualization.

9.3 Evaluation

To validate the visualization model that we propose in this thesis, we implemented it in three applications for assessment of software evolution, described in Chapter 5, 6 and 7. We used these applications to perform a number of informal user experiments on real life evolution data and several types of users.

The first experiment we organized was a user study in which software developers used the visualization proposed in Chapter 5 to get insight in the structure and evolution of unknown source code files. A small number of developers took part in the study. Each developer was assessed by a domain expert, during a 15 minute test, after receiving a 15 minute training session on using the tool. At the end of the test, the domain experts were asked to validate the user findings and to reason about the quality of the acquired insight. The outcome of this study signaled the potential of the proposed visualization for supporting software developers in during the maintenance phase of software projects. Yet this study has a couple of issues that must be considered when assessing the validity of the results:

- It is not known whether the training period and the test duration are adequately chosen for the complexity of the tool. A more thorough user study would be necessary first to identify this aspect.
- The participation of the study subjects took place on a voluntary, non-compensatory basis. The lack of a drive for performing investigations might bias the study results, as it doesn't address precisely the target audience of the tools. It is not known, however, what the biasing level might be in this case.
- The study cannot be considered statistically relevant. It contained only a relatively small number of sessions in which the findings of one user, assessing the evolution

of one file from one project, were observed by one domain expert. To obtain statistical relevance this has to be ensured for every above mentioned parameter, that is for: users, files, projects, and domain experts. Yet, the organization of such a study in a supervised way requires a large amount of resources.

Given the above issues of organizing relevant user studies we tried to adopt other means for validating our visualizations. One useful alternative we found was to take part in open contests for investigating software evolution (*e.g.*, the MSR Challenge of MSR'06 [116], the Tool Challenge of SoftVis'06 [115] and VisSoft'07 [117]). This enabled us to demonstrate the use of our visualizations to a jury of domain experts, achieving statistical relevance in this particular respect. We used this approach for validating the visualization proposed in Chapter 6.

Another alternative we used in our experiments in Chapters 6 and 7 was to ask domain and tool experts to identify useful usage scenarios for the tools we implemented. These included senior software developers, architects and project managers from a large software intensive products company and senior contributors from the KDE [66] Open Source project. While this approach does not offer a proper validation of the usefulness of the proposed model for assessing software evolution, it gives examples on the type of insight that can be obtained, and validates the model for specific investigations.

One possible solution for addressing all above mentioned challenges of organizing relevant user studies is the involvement of the Open Source Software community. We did not investigate this alternative yet, but it seems promising as it addresses all presented issues. One possible drawback we foresee with such an approach, however, is the difficulty of interpreting negative results. While the acceptance of a tool by the community would be a statistical relevant validation of the visualization model that the tool proposes, the lack of reaction would not reveal anything in this respect. Additionally, this approach requires considerable resources to be invested in tool maintenance, documentation and dissemination, in order to make acceptance possible at all.

Chapter 10

Conclusions

In this thesis, we tried to find a visual approach to answer the question “How to enable users to get insight in the evolution of a software system?”. Answers to this question can bring insight in the software development context and its evolution trends, and can improve both software understanding and decision making during the maintenance phase of large projects.

To this end, we proposed a model for software evolution visualization. This model is based on the assumption that developers are familiar with code representations similar to the ones they use for building the code. This model can show insertion, deletion and continuation evolution patterns for common representations of software, such as file as a set of lines, project as a set of files, project as software unit. The model builds on the standard visualization pipeline and gives a number of steps and guidelines to be followed when designing visualizations of software evolution.

We have presented three implementations of this model for visualizing evolution of software based on information extracted from Software Configuration Management (SCM) repositories, at three different levels of detail, and covering different types of tasks. In this chapter, we present the concluding remarks of our work. These are meant both for engineers interested in implementing the model in real life applications and for researchers that would like to investigate the limitations of the proposed approach and look for possible solutions.

10.1 On Data Preprocessing

SCM repositories are the only available instruments at the moment that store the intermediate states a software system has during its life cycle. However, SCM systems have not been specifically designed to record the evolution of software. First of all, not all evolution patterns can be directly retrieved from such repositories. Instead, they have to be detected via supplementary analyses. Secondly, only few representations of software are supported by the existing SCMs. To assess software evolution from the perspective

of other representations, one has to recover these representations and the associated evolution patterns based on the information stored on SCMs. Both tasks are acknowledged as difficult by the research community, and they need further investigation before one can implement them in practice. Ultimately, this may lead to the advent of a new generation of SCM tools.

10.2 On Software Evolution Visualization

From a visualization point of view, software evolution assessment is a difficult problem because the evolution data is abstract, highly variate and the amount of data is very large. When addressing the challenge of building a visualization of software evolution one could benefit from the experience findings we detail in Chapter 9.

The visualization model that we offer can be used to implement practical assessment tools for software evolution, the indented audience ranging from software developers to architects and project managers. The use cases we presented in this thesis cover different usage scenarios from familiarizing with the structures and issues in a new project, to detecting the knowledge distribution of the team and identifying complexity trends. Software evolution assessment tools are highly exploratory in nature. Consequently, two aspects have to be considered when trying to build an efficient implementation. First of all, interaction is critical for enabling the user to steer the investigation process. Secondly, multiple levels of software representation detail should be addressed from within the same application. These would enable users to initiate investigations at a higher abstraction level and then conduct analysis drills when interesting events are detected. Conversely, investigations performed at a high level of detail can be reported to the context offered by higher abstraction levels. Both ways of navigating the abstraction scale are important for usable implementations of our model in practice.

When building our visualizations of software evolution we started from the assumption that users are familiar with software representations that are close to the ones they use to construct it. Consequently, our visualizations present software at the level of lines of source code, files and system. One common representation of software that we did not address in our investigation is software as a collection of interrelated classes. This case could be treated similarly to the file level representation of software. Nevertheless, our visualization model does not support the visualization of class relations. If these are important to the investigation, our model must be augmented accordingly, or another visualization model has to be found.

10.3 On Evaluation

To evaluate the visualization model we propose in this thesis, we organized a number of experiments using particular implementations of our model.

One experiment took the form of an informal user study, performed in the presence of a domain expert (Chapter 5). Three experiments consisted of participation in tool demos and challenges organized in conjunction with specialized workshops and symposia in the

field of software visualization and software repositories mining (see [115, 116, 117]). Two other experiments had the form of illustrative usage scenarios performed by domain experts on real life software projects (Chapters 6 and 7). All experiments confirmed the potential of the visualization model we propose for the assessment of software evolution.

However, the results of these evaluation experiments are far from statistically relevant. Organizing such a broad experiment is difficult for several reasons. First of all, the intended audience for our visual tools is formed of relatively highly qualified users, with a good knowledge and experience in software development. It is difficult to involve a large number of such users in an evaluation study. Additionally, users want to try our tools each for their own projects, which they know well, and are less interested in exploring a test project.

Secondly, the tasks that users could perform using the proposed visualization model cover a wide range of activities and are based on a wide number of parameters. Key in this process is obtaining insight, and, as noted by North [88], this is notoriously hard to measure and evaluate. There is no formal model yet to assess the relevance and the completeness of this set of tasks. Evaluation has to be done, therefore, by a domain expert who can only deliver an informal assessment of the user performance. Consequently, a large number of domain experts is required to achieve statistical relevance. Once more, this is very difficult to achieve in a controlled way and in a limited time period.

One way that we can imagine for achieving statistical relevance is to make use of the Open Source community as an unsupervised engine for evaluating software engineering tools. In this case, the emphasis of a possibly relevant user study would be on advertising a tool implementation of the model inside the community. If the tool becomes popular and gets accepted, the visualization model gets validated. Time could be an issue in this respect. However, if the tool does not become popular, which can be due to secondary reasons such as lack of support, documentation or publicity, the study becomes irrelevant, and the evaluation of the model remains undecided.

10.4 Future Work

The visualization model we propose in this thesis can be used to investigate software evolution related aspects for some commonly used software representations. The model is nevertheless not complete in this respect, and can be extended in a number of directions.

First of all, our model addresses only insertion, continuation and deletion evolution patterns. There are interesting scenarios for assessing the evolution of software systems targeting split and merge patterns. Future work could try to extend our model in this direction. This could enable users to identify moments of refactoring / rearchitecting in the evolution of a software system.

Furthermore, software has many possible representations. The use of a given representation in practice depends on many factors, which are project and context specific. Implementations of the proposed model to visualize evolution for other software representations than the one presented in Section 3.3, and using data from other SCM repositories than CVS and Subversion would be necessary. These can bring new insight about the

usability of the model, and the challenges of implementing it in practice.

One specific difficult problem we foresee in this direction is the implementation of our model for software representations that rely heavily on relations between entities (*e.g.*, class diagrams). To perform meaningful investigations in such cases, additional methods and techniques are required to visually encode relations. This might require augmenting the model we propose, or replacing it entirely with a new one.

Another important issue that is yet to be solved when assessing the evolution of software is the availability of data. SCM repositories are the only tools available at this moment for recording software evolution. Yet they are designed for another purpose, namely to support distributed development. Consequently, SCMs neglect important information (*e.g.*, entity merge events), which makes recovery of evolution data difficult. Therefore, a new approach for recording and storing the evolution of software is required. This should be specifically designed to:

- make a wider range of evolution patterns available for investigation by recording split and merge patterns for all covered software entities in all representations;
- cover a wider range of software representations, such as UML designs and abstract syntax trees;
- integrate with other project support systems (*e.g.*, documentation, bug tracking, quality monitoring) to enable correlation discovery across a larger set of software related entities.

Last but not least, the model we propose cannot be easily evaluated in a supervised way, given the specialized audience it targets: the software engineering community. However, some representative members of this audience are part of an organization that might facilitate evaluation studies in an unsupervised way: the Open Source Software community. A method for increasing the chances of performing relevant studies using this community is necessary. Such a method would be useful for evaluating not only our visualization model but any software engineering related tool. After all, the Open Source Software forums are a platform for introducing and validating software applications in any domain.

We have made our software available to the Open Source Software community (see [29]). With this we hope to obtain feedback on all aspects of our methods and techniques. Additionally, we hope to have contributed to the software engineering field in general, by enabling developers and maintainers to obtain insight in the history of large software projects in an effective and efficient way.

Bibliography

- [1] ABELLO, J., AND VAN HAM, F. Matrix zoom: A visual interface to semi-external graphs. In *InfoVis'04: Proceedings of the 10th IEEE Symposium on Information Visualization* (2004), IEEE Computer Society, pp. 183–190.
- [2] AGE OF EMPIRES, online. www.ageofempires.com.
- [3] ANTLR, online. www.antlr.org.
- [4] AUBER, D. Tulip. *Lecture Notes in Computer Science* 2265 (2001), 335–337.
- [5] BALL, T., KIM, J., PORTER, A., AND SIY, H. If your version control system could talk... In *ICSE'97: Proceedings of Workshop on Process Modeling and Empirical Studies of Software Engineering* (1997).
- [6] BASILI, V., AND MILLS, H. Understanding and documenting programs. *IEEE Transactions on Software Engineering* SE-8 (May 1982), 270–283.
- [7] BASSIL, B., AND KELLER, R. Software visualization tools: Survey and analysis. In *IWPC'01: Proceedings of the 9th International Workshop on Program Comprehension* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 7–17.
- [8] BECK, K., AND ANDRES, C. *Extreme Programming Explained: Embrace Change*, second ed. Addison-Wesley, 2004.
- [9] BECKER, R., EICK, S., AND WILKS, A. Visualizing network data. *IEEE TVCG* 1, 1 (March 1995), 16–28.
- [10] BENNET, P. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.
- [11] BIEMAN, J., ANDREWS, A., AND YANG, H. Understanding change-proneness in oo software through visualization. In *IWPC'03: Proceedings 11th International Workshop on Program Comprehension* (2003), IEEE CS Press, pp. 44–53.
- [12] BIGGERSTAFF, T., MITBANDER, B., AND WEBSTER, D. Program understanding and the concept assignment problem. *Commun. ACM* 37, 5 (1994), 72–82.
- [13] BONSAI, online. www.mozilla.org/projects/bonsai/.

- [14] BOOCH, G., online. [www.booch.com/architecture/blog/artifacts/Software Architecture.ppt](http://www.booch.com/architecture/blog/artifacts/Software%20Architecture.ppt).
- [15] BURCH, M., DIEHL, S., AND WEISSGERBER, P. Visual data mining in software archives. In *SoftVis'05: Proceedings of the 2005 ACM Symposium on Software Visualization* (New York, NY, USA, 2005), ACM Press, pp. 37–46.
- [16] BURROWS, C., AND WESLEY, I. *Ovum evaluates: configuration management*. Ovum Inc., 1999.
- [17] CARD, S., MACKINLAY, J., AND SCHNEIDERMAN, B. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.
- [18] CHIKOFFSKY, E., AND CROSS, J. I. Reverse engineering and design recovery: a taxonomy. *IEEE Software* 7, 1 (January 1990), 13–17.
- [19] CLEARCASE, online. www-306.ibm.com/software/awdtools/clearcase/.
- [20] CM SYNERGY, online. www.telelogic.com.
- [21] COLEMAN, D.M. ASH, D., LOWTHER, B., AND OMAN, P. Using metrics to evaluate software system maintainability. *IEEE Computer* 27, 8 (1994), 44–49.
- [22] COLLBERG, C., KOBOUROV, S., NAGRA, J., PITTS, J., AND WAMPLER, K. A system for graph-based visualization of the evolution of software. In *SoftVis'03: Proceedings of the 2003 ACM Symposium on Software Visualization* (2003), ACM Press, pp. 77–86.
- [23] COLUMBUS, online. www.frontendart.com.
- [24] CORBI, T. Program understanding: challenge for the 1990's. *IBM Syst. J.* 28, 2 (1989), 294–306.
- [25] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to Algorithms*, 16th ed. MIT Press, 1996.
- [26] COVERITY, online. www.coverity.com.
- [27] CUBRANIC, D., MURPHY, G., SINGER, J., AND BOOTH, K. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering* 31, 6 (2005), 446–465.
- [28] CVS, online. www.nongnu.org/cvs/.
- [29] CVSGRAB, online. www.win.tue.nl/vis/.
- [30] CVSSCAN, online. www.win.tue.nl/vis/.
- [31] DIEHL, S. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.
- [32] DIFF, online. www.gnu.org/software/diffutils/diffutils.html.

- [33] DUCASSE, S., GARBA, T., AND NIERSTRASZ, O. Moose: an agile reengineering environment. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference* (New York, NY, USA, 2005), ACM Press, pp. 99–102.
- [34] ECLIPSE CVS, online. drupal.org.
- [35] ED2K, online. www.edonkey2000.com.
- [36] EICK, S. Aspects of network visualization. *IEEE Computer Graphics & Applications* 16, 2 (March 1996), 69–72.
- [37] EICK, S. G., STEFFEN, J., AND SUMNER, E. Seesoft – a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering* 18, 11 (1992), 957–968.
- [38] ELRAD, T., FILMAN, R., AND BADER, A. Aspect-oriented programming. *CACM: Communications of the ACM* 44 (2001).
- [39] eMULE, online. www.emule-project.net.
- [40] ERLIKH, L. Leveraging legacy system dollars for e-business. (*IEEE*) *IT Pro* (May-June 2000), 17–23.
- [41] EVERITT, E., LANDAU, S., AND LEESE, M. *Cluster Analysis*. Arnold Publishers, 2001.
- [42] EZEL, online. www.win.tue.nl/vis/.
- [43] FENTON, N., AND PFLEEGER, S. *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997.
- [44] FEWSTER, M., AND GRAHAM, D. *Software Test Automation: Effective Use of Test Execution Tools*, 1st ed. Addison-Wesley Professional, 1999.
- [45] FISCHER, M., PINZGER, M., AND GALL, H. Populating a release history database from version control and bug tracking systems. In *ICSM'03: Proceedings of the 19th International Conference on Software Maintenance* (2003), IEEE CS Press, pp. 23–32.
- [46] FREEDESKTOP, online. www.freedesktop.org.
- [47] FROELICH, J., AND DOURISH, P. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering* (2004), IEEE CS Press, pp. 387–396.
- [48] GALL, H., JAZAYERI, M., AND KRAJEWSKI, J. Cvs release history data for detecting logical couplings. In *IWPSE'03: Proceeding of the 6th International Workshop on Principles of Software Evolution* (2003), IEEE CS Press, pp. 13–23.
- [49] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- [50] GANTT, H. Work, wages and profits. *The Engineering Magazine* (1910), 312.
- [51] GERMAN, D., HINDLE, A., AND JORDAN, N. Visualizing the evolution of software using softchange. In *SEKE'04: Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering* (2004), pp. 336–341.
- [52] GERMAN, D., AND MOCKUS, A. Automating the measurement of open source projects. In *OOSE'03: Proceeding of Workshop on Open Source Software Engineering* (2003).
- [53] GERSHON, N. Information visualization: The next frontier. In *SIGGRAPH'94: Proceedings of International Conference and Exhibition on Computer Graphics and Interactive Techniques* (1994), pp. 485–486.
- [54] GNU RCS, online. www.gnu.org/software/rcs/rcs.html.
- [55] GODFREY, M., AND ZOU, L. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering* 31, 2 (February 2005), 166–181.
- [56] GRISWOLD, W., YUAN, J., AND KATO, Y. Exploiting the map metaphor in a tool for software evolution. In *ICSE'01: Proceedings of the 23rd International Conference on Software Engineering* (2001), IEEE CS Press, pp. 265–274.
- [57] HANNEMANN, J., AND KICZALES, G. Design pattern implementation in java and aspectj. In *OOPSLA'02: Proceedings of the Annual ACM SIGPLAN Conferences on Object-Oriented Programming* (2002), ACM Press, pp. 161–173.
- [58] HAVRE, S., HETZLER, B., AND NOWELL, L. Themeriver: Visualizing theme changes over time. In *InfoVis'00: Proceedings of the 6th IEEE Symposium on Information Visualization* (2000), IEEE Computer Society, pp. 115–123.
- [59] HISTORIAN, online. historian.tigris.org.
- [60] HOLTEN, D., VLIEGEN, R., AND VAN WIJK, J. Visual realism for the visualization of software metrics. In *VISSOFT'05: Proceedings of 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2005), IEEE CS Press, pp. 27–32.
- [61] HUNT, J., AND TICHY, W. Extensible language-aware merging. In *ICSM'03: Proceedings of the 19th IEEE International Conference on Software Maintenance* (2002), IEEE CS Press, pp. 511–520.
- [62] ITK, online. www.itk.org.
- [63] JAVACVS, online. javacvs.sourceforge.net.
- [64] JERDING, D., AND STASKO, J. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics* 4, 3 (1998), 257–271.
- [65] KAN, S. *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional, 2002.

- [66] KDE KOFFICE, online. www.koffice.org.
- [67] KOIKE, H., AND CHU, H.-C. Vrcs: integrating version control and module management using interactive three-dimensional graphics. In *VL'97: Proceedings of the 1997 IEEE Symposium on Visual Languages* (1997), IEEE CS Press, pp. 168–173.
- [68] KOSCHKE, R. Software visualization for reverse engineering. In *Revised Lectures on Software Visualization, International Seminar* (2002), Springer-Verlag, pp. 138–150.
- [69] LANZA, M. The evolution matrix: recovering software evolution using software visualization techniques. In *IWPSE'01: Proceedings of the 4th International Workshop on Principles of Software Evolution* (2001), ACM Press, pp. 37–42.
- [70] LEHMAN, M. Laws of software evolution revisited. In *EWSPT'96: Proceedings of the 5th European Workshop on Software Process Technology* (London, UK, 1996), Springer-Verlag, pp. 108–124.
- [71] LIBCVS, online. www.gnu.org/software/libcvs-spec/.
- [72] LOMMERSE, G., NOSSIN, F., VOINEA, L., AND TELEA, A. The visual code navigator: An interactive toolset for source code investigation. In *InfoVis'05: Proceedings of the 11th IEEE Symposium on Information Visualization* (2005), IEEE Computer Society, pp. 24–31.
- [73] LOPEZ-FERNANDEZ, L., ROBLES, G., AND GONZALEZ-BARAHONA, J. Applying social network analysis to the information in cvs repositories. In *MSR'04: Proceedings of the 2004 International Workshop on Mining Software Repositories* (2004).
- [74] MAGNAVIEW, online. www.magnaview.nl.
- [75] MALETIC, J., MARCUS, A., AND FENG, L. Source viewer 3d (sv3d) - a framework for software visualization. In *ICSE'03: Proceedings of the 25th International Conference on Software Engineering* (2003), IEEE CS Press, pp. 812–813.
- [76] MARSHALL, M., HERMAN, I., AND MELANÇON, G. An object-oriented design for graph visualization. *Software: Practice and Experience* 31, 8 (2001).
- [77] MATKOVIC, K., HAUSER, H., SAINTITZER, R., AND GRÖLLER, M. Process visualization with levels of detail. In *InfoVis'02: Proceedings of the 8th IEEE Symposium on Information Visualization* (2002), pp. 67–70.
- [78] MAZZA, C. ESA software engineering standards. Tech. rep.
- [79] MCCABE, T., AND WATSON, A. Software complexity. *Crosstalk, Journal of Defense Software Engineering* 7, 12 (December 1994), 5–9.
- [80] MCCLAVE, J., AND SINCICH, T. *Statistics*, 10th ed. Prentice Hall, 2006.
- [81] MCRL2, online. www.mcrl2.org.

- [82] MOCKUS, A., FIELDING, R., AND HERBSLEB, J. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology* 3, 11 (2002).
- [83] MORETA, S., AND TELEA, A. Multiscale visualization of dynamic software logs. In *EUROVIS'05: Proceedings of the 2005 IEEE Eurographics Symposium on Visualization* (2007), IEEE CS Press.
- [84] MORETA, S., AND TELEA, A. Visualizing dynamic memory allocations. In *VIS-SOFT'07: Proceedings of 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (to appear)* (2007), IEEE CS Press.
- [85] MUNZNER, T., GUIMBRETIERE, F., TASIRAN, S., ZHANG, L., AND ZHOU, Y. Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Trans. Graph.* 22, 3 (2003), 453–462.
- [86] NAUR, P., AND RANDELL, B. *Software Engineering. Report of a conference sponsored by the NATO Science Committee.* NATO Scientific Affairs Division, 1969.
- [87] NETBEANS.JAVACVS, online. javacvs.netbeans.org.
- [88] NORTH, C. Toward measuring visualization insight. *IEEE Computer Graphics & Applications* 26, 3 (2006), 6–9.
- [89] OHIRA, M., OHSUGI, N., OHOKA, T., AND MATSUMOTO, K. Accelerating cross project knowledge collaboration using collaborative filtering and social networks. In *MSR'05: Proceedings of 2005 International Workshop on Mining Software Repositories* (2005), pp. 111–115.
- [90] PAUL, S., PRAKASH, A., BUSS, E., AND HENSHAW, J. Theories and techniques of program understanding. In *CASCON'91: Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research* (1991), IBM Press, pp. 37–53.
- [91] PINZGER, M., GALL, H., FISCHER, M., AND LANZA, M. Visualizing multiple evolution metrics. In *SoftVis'05: Proceedings of the 2005 ACM Symposium on Software Visualization* (New York, NY, USA, 2005), ACM Press, pp. 67–75.
- [92] RAO, R., AND CARD, S. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *CHI'94: Proceedings of the 1994 SIGCHI Conference on Human Factors in Computing Systems* (1994), ACM Press, pp. 318–322.
- [93] RIVA, C. Visualizing software release histories with 3dsoftvis. In *ICSE'00: Proceedings of the 22nd International Conference on Software Engineering* (New York, NY, USA, 2000), ACM Press, p. 789.
- [94] SAITO, T., MIYAMURA, H., YAMAMOTO, M., SAITO, H., HOSHIYA, Y., AND KASEDA, T. Two-tone pseudo coloring: Compact visualization for one-dimensional data. In *InfoVis'05: Proceedings of the Proceedings of the 11th IEEE Symposium on Information Visualization* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 173–180.

- [95] SEO, J., AND SHNEIDERMAN, B. Interactively exploring hierarchical clustering results. *Computer* 35, 7 (2002), 80–86.
- [96] SHNEIDERMAN, B. The eyes have it: A task by data type taxonomy for information visualization. In *VL'96: Proceedings of the 1996 IEEE Symposium on Visual Languages* (1996), IEEE CS Press, pp. 336–343.
- [97] SLIWERSKI, J., ZIMMERMANN, T., AND ZELLER, A. When do changes induce fixes? on fridays. In *MSR'05: Proceedings of the 2005 International Workshop on Mining Software Repositories* (2005), pp. 24–28.
- [98] SPENCE, R. *Information Visualization*. ACM Press, 2001.
- [99] SQLITE, online. www.sqlite.org.
- [100] STANDISH, T. An essay on software reuse. *IEEE Transactions on Software Engineering* 10, 5 (September 1984), 494–497.
- [101] STASKO, J., DOMINGUE, J., BROWN, M., AND PRICE, B. *Software Visualization*. MIT Press, 1998.
- [102] STOREY, M., WONG, K., AND MULLER, H.
- [103] STROUSTRUP, B. *The C++ Programming Language (3rd edition)*. Addison Wesley Longman, 1997.
- [104] SVN, online. subversion.tigris.org.
- [105] TELEA, A., MACCARI, A., AND RIVA, C. An open toolkit for prototyping reverse engineering visualization. In *VisSym'02: Proceedings of the 2002 Joint Eurographics - IEEE TCVG Symposium on Visualization* (2002), The Eurographics Association, pp. 241–251.
- [106] TERMEER, M., LANGE, C., TELEA, A., AND CHAUDRON, M. Visual exploration of combined architectural and metric information. In *VISSOFT'05: Proceedings of 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2005), IEEE CS Press, pp. 21–26.
- [107] THOMAS, J., AND COOK, K. A visual analytics agenda. *Computer Graphics and Applications* 26, 1 (Jan. - Feb. 2006).
- [108] TILLEY, S., WONG, K., STOREY, M., AND MULLER, H. Rigi: A visual tool for understanding legacy systems. *International Journal of Software Engineering and Knowledge Engineering* (December 1994).
- [109] TONELLA, P., AND POTRICH, A. *Reverse Engineering of Object Oriented Code*. Springer, 2004.
- [110] TRETMANS, G., AND BELINFANTE, A. *Automatic Testing with Formal Methods*. University of Twente, The Netherlands, 2000.
- [111] VAN WIJK, J., AND VAN DE WETERING, H. Cushion treemaps: Visualization of hierarchical information. In *InfoVis'99: Proceedings of the 5th IEEE Symposium on Information Visualization* (1999), IEEE Computer Society, pp. 73–78.

- [112] VAN WIJK, J., AND VAN OVERVELD, C. Preset based interaction with high dimensional parameter spaces. In *Proceedings of the 1999 Dagstuhl Seminar on Visualization* (1999).
- [113] VAN WIJK, J., AND VAN SELOW, E. Cluster and calendar based visualization of time series data. In *InfoVis'99: Proceedings of the 5th IEEE Symposium on Information Visualization* (1999), IEEE Computer Society, pp. 4–9.
- [114] VISUAL SOURCESAFE, online. msdn.microsoft.com/ssafe/.
- [115] VOINEA, S., AND TELEA, A. How do changes in buggy mozilla files propagate? In *SoftVis'06: Proceedings of the 2006 ACM Symposium on Software Visualization* (2006), ACM Press, pp. 147–148.
- [116] VOINEA, S., AND TELEA, A. Mining software repositories with cvsgrab. In *MSR'06: Proceedings of the 2006 International Workshop on Mining Software Repositories* (2006), pp. 167–168.
- [117] VOINEA, S., AND TELEA, A. Visualizing debugging activity in source code repositories. In *VISSOFT'07: Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (to appear)* (2007), IEEE CS Press.
- [118] VTK, online. www.vtk.org.
- [119] WARDEN, R. *Software Reuse and Reverse Engineering in Practice*. Chapman & Hall, 1992.
- [120] WARE, C. *Information Visualization. Perception for Design*. Morgan Kaufmann, 2000.
- [121] WARE, C. Designing with a 2 1/2d attitude. *Information Design Journal* 3, 10 (2001).
- [122] WILLS, L., AND NEWCOMB, P. *Reverse Engineering*. Springer, 2001.
- [123] WINCVS, online. www.wincvs.org.
- [124] WINDIFF, online. www.microsoft.com.
- [125] WU, J., SPITZER, C., HASSAN, A., AND HOLT, R. Evolution spectrographs: Visualizing punctuated change in software evolution. In *IWPSE'04: Proceedings of the 7th International Workshop on Principles of Software Evolution* (2004), IEEE CS, pp. 57–66.
- [126] WU, X. *Visualization of version control information. Master's thesis*. University of Victoria, 2003.
- [127] YEE, K.-P., FISHER, D., DHAMIJA, R., AND HEARST, M. Animated exploration of dynamic graphs with radial layout. In *InfoVis'01: Proceedings of 7th IEEE Symposium on Information Visualization* (2001), pp. 43–50.

- [128] YING, A., MURPHY, G., NG, R., AND CHU-CARROLL, M. Predicting source code changes by mining revision history. *IEEE Transactions on Software Engineering* 30, 9 (September 2004).
- [129] ZIMMERMANN, T., DIEHL, S., AND ZELLER, A. How history justifies system architecture (or not). In *IWPSE'03: Proceedings of the 6th International Workshop on Principles of Software Evolution* (2003), IEEE Computer Society, pp. 73–83.
- [130] ZIMMERMANN, T., AND WEISSGERBER, P. Preprocessing cvs data for fine-grained analysis. In *MSR'04: Proceedings of the 2004 International Workshop on Mining Software Repositories* (2004).
- [131] ZIMMERMANN, T., WEISGERBER, P., DIEHL, S., AND ZELLER, A. Mining version histories to guide software changes. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering* (2004), IEEE Computer Society, pp. 563–572.

Publications related to this work

- VOINEA, L., LUKKIEN, J.J. AND TELEA, A., Visual Assessment of Software Evolution, in *Science of Computer Programming*, vol. 65, no. 3, Elsevier, 2007, pp. 222–248
Related chapter: 5
- VOINEA, L., AND TELEA, A., Visual Querying and Analysis of Large Software Repositories, accepted for *Special Issue on Mining Software Repositories of Empirical Software Engineering*, Springer, 2007, to appear
Related chapter: 6
- VOINEA, L., AND TELEA, A., Visualizing Debugging Activity in Source Code Repositories, in *VISSOFT'07: Proceedings of 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, IEEE Press, 2007, to appear
Related chapter: 6
- VOINEA, L., AND TELEA, A., Visual Data Mining and Analysis of Software Repositories, accepted for *Special Issue on Visual Analytics of Computers & Graphics*, Elsevier, 2006, to appear
Related chapters: 2, 4, 5, 6
- VOINEA, L. AND TELEA, A., A File-Based Visualization of Software Evolution, in *ASCI'06: Proceedings of the Annual Conference of the Advanced School for Computing and Imaging*, ASCI Press, 2006, pp. 114–121
Related chapter: 6
- VOINEA, L. AND TELEA, A., How Do Changes in Buggy Mozilla Files Propagate?, in *SoftVis'06: Proceedings of the 2006 ACM Symposium on Software Visualization*, ACM Press, 2006, pp. 147–148
Related chapter: 6
- VOINEA, L. AND TELEA, A., Multiscale and Multivariate Visualizations of Software Evolution, in *SoftVis'06: Proceedings of the 2006 ACM Symposium on Software Visualization*, ACM Press, 2006, pp. 115–124, (Cover image)
Related chapter: 6
- VOINEA, L. AND TELEA, A., Mining Software Repositories with CVSgrab, in *MSR'06: Proceedings of the 2006 International Workshop on Mining Software*

Repositories, ACM Press, 2006, pp. 167–168, (Second prize at the MSR Challenge competition)

Related chapter: 6

- VOINEA, L. AND TELEA, A., An Open Framework for CVS Repository Querying, Analysis and Visualization, in *MSR'06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, ACM Press, 2006, pp. 33–39
Related chapter: 6
- VOINEA, L. AND TELEA, A., CVSgrab: Mining the History of Large Software Projects, in *EUROVIS'06: Proceedings of the 2006 Eurographics / IEEE VGTC Symposium on Visualization*, IEEE Press, 2006, pp. 187–194
Related chapter: 6
- VOINEA, L. AND TELEA, A., Interactive Visual Mechanisms for Exploring Source Code Evolution, in *VISSOFT'05: Proceedings of 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, IEEE Press, 2007, IEEE CS Press, pp. 52–57
Related chapter: 5
- LOMMERSE, G., NOSSIN, F., VOINEA, L. AND TELEA, A., The Visual Code Navigator: An Interactive Toolset for Source Code Investigation, in *InfoVis'05: Proceedings of the 11th IEEE Symposium on Information Visualization*, IEEE CS Press, pp. 24–31
Related chapter: 5
- VOINEA, L. AND TELEA, A., Visual Assessment Techniques for Component-Based Framework Evolution, in *EUROMICRO'05: Proceedings of the 31st EUROMICRO Conference*, IEEE CS Press, pp. 168–179
Related chapter: 5
- VOINEA, L., TELEA, A. AND VAN WIJK, J.J., A Line-Based Visualization of Code Evolution, in *ASCI'05: Proceedings of the Annual Conference of the Advanced School for Computing and Imaging*, ASCI Press, 2005, pp. 350–357
Related chapter: 5
- VOINEA, L., TELEA, A. AND CHAUDRON, M.R.V., Version-Centric Visualization of Code Evolution, in *EUROVIS'05: Proceedings of the 2005 Eurographics / IEEE VGTC Symposium on Visualization*, IEEE CS Press, pp. 223–230
Related chapter: 5
- VOINEA, L., TELEA, A., and Van Wijk, J.J., CVSscan: Visualization of Code Evolution, in *SoftVis'05: Proceedings of the 2005 ACM Symposium on Software Visualization*, ACM Press, pp. 47–56
Related chapter: 5
- VOINEA, L., TELEA, A., AND VAN WIJK, J.J., A Visual Assessment Tool for P2P File Sharing Networks, in *InfoVis'04: Proceedings of the 10th IEEE Symposium on Information Visualization*, IEEE CS Press, 2004, pp. 41–48, (Cover image)
Related chapter: 8

- VOINEA, L. AND TELEA, A., A Framework for Interactive Visualization of Component-Based Software, in *EUROMICRO'04: Proceedings of the 30st EUROMICRO Conference*, IEEE CS Press, pp. 567–574
Related chapter: 5

Summary

Software has today a large penetration in all infrastructure levels of the society. This penetration took place rapidly in the last two decades and continues to increase. In the same time, however, the software industry gets confronted with two increasingly serious challenges: complexity and evolution. The size of software applications is growing larger. This leads to a steep increase in complexity. Additionally, the change in requirements and available technologies leads to software modifications. As a result, a huge amount of code needs to be maintained and updated every year (i.e. the legacy systems problem).

Software visualization is a very promising solution to the above mentioned challenges of the software industry. It is a specialized branch of information visualization, which visualizes artifacts related to software and its development process. In this thesis we try to use visualization of software evolution to get insight in the development context of software and in its evolution trends. The main question we try to answer with this is:

“How to enable users to get insight in the evolution of a software system?”

Our final goal is to improve both software understanding and decision making during the maintenance phase of large software projects.

We start by positioning the thesis in the context of related research in the field of both software evolution analysis and visualization. Then we perform an analysis of the software evolution domain to formalize the problems specific to this field. To this end, we propose a generic system evolution model and a structure based meta-model for software description. Consequently, we use these models to give a formal definition of software evolution.

Next we propose a visualization model for software evolution, based on the previously introduced software evolution model. The visualization model consists of a number of steps with specific guidelines for building visual representations. Then we present three applications that make use of the proposed visualization model to support real life software evolution analysis scenarios. These applications cover the most commonly used software description models in industry: file as a set of code lines, project as a set of files, and project as a unitary entity. For each application, we formulate relevant use cases, present specific implementation aspects, and discuss results of use case evaluation experiments.

We also propose in this thesis a novel visualization of data exchange processes in

Peer-to-Peer networks. While this does not address software evolution, it tackles comparable issues, e.g., the dynamic evolution of a set of interrelated data artifacts. The aim of presenting this visualization is twofold. First, we illustrate how to visualize different types of software-related data than purely software source code. Secondly, we show that the visual and interaction techniques that we have developed in the context of software evolution visualization can be put to a good use for other applications as well.

We conclude the thesis with an inventory of reoccurring problems and solutions we have discovered in the visualization of software evolution. Additionally, we identify generic issues that transcend the border of the software evolution domain and we present them together with a set of recommendation for their broader applicability. Finally, we outline the remaining open issues, and the possible research directions that can be followed to address them.

Acknowledgements

Throughout this thesis I have used the pronoun “we” instead of “I” when I referred to the work that was done. However, “we” is far from being the *pluralis majestatis*. It has a more humble meaning. It acknowledges that the work presented in the thesis is based on the efforts of more people than just me, people to whom I am very thankful.

First of all, I owe a great deal of gratitude to dr.ir. Alex Telea, my first copromotor and direct supervisor, who closely supported me during the four years I spent on this thesis. He always helped me find the path I was looking for, and supplied me with the necessary motivation to carry on when wandering in the endless mazes of visualization and software engineering.

I would like to thank prof.dr.ir. Jack van Wijk, my promotor. His critical and constructive observations helped me to maintain an objective stance throughout my research, keeping a balance between my enthusiasm and my claims.

I am also grateful to dr. Johan Lukkien, my second copromotor, for challenging me to look at my work from different perspectives and for helping me to perfect my models of software evolution.

I am very thankful to the members of my doctoral committee, prof.dr. Stephan Diehl, prof.dr. Mark van den Brand, prof.dr. Arie van Deursen, and prof.dr. Jos Roerdink for their constructive remarks on my thesis manuscript and for accepting to be part of the opposition.

My roommates have also had an important contribution to this thesis. In chronological order of meeting them, I would like to thank Hannes Pretorius, Frank van Ham, Dennie Reniers and Danny Holten, Koray Duhbaci, Yedendra Shrinivasan, and Jing Li for their suggestions, help and the interesting discussions we had over lunch or coffee. They knew how to make our big room a welcoming place that I was always entering with pleasure.

I would also like to thank my other colleagues from the VIS, OAS and SAN groups, for offering me a challenging working environment and a lot of inspiration during the last four years.

I am equally grateful to Tineke van den Bosch, Cecile Brouwer, and all the employees of the PO department that took care of the administrative jobs and helped me fight the bureaucracy that comes with being a foreigner in The Netherlands. Their faithful support allowed me to concentrate on my work and worry less about administrative problems.

I would like to give special thanks to Harold Weffers and Maggy de Wert from the Stan Ackermans Institute for being very supportive both from a professional and personal point of view. They have always welcomed me and made me feel appreciated throughout my staying at Technische Universiteit Eindhoven.

On the personal side, family support is something I could have not worked without. First of all I would like to thank my parents for their unconditional love and support. They had a great influence in me taking up the challenge of a doctorate. I thank my mother, Elena, for making me appreciate and respect the world of science, and my father, Radu, for giving me the sense of adventure and the curiosity with respect to foreign cultures.

I would especially like to thank to my grandmother, Mami, for taking care of me for such a long time, and for teaching me to hope and to have faith, qualities that are so necessary when you embark on an unknown journey.

I am enormously grateful to my close relatives, Jeni, Bazil, Radu, George, Alina and Sorin for taking care of the family I left behind when moving to The Netherlands. Without their support I could have not concentrated on my work, far away from home.

Last but not least, I would like to thank my wife, Daniela. Her love and support have been the only solid ground I had all these years. *Iti multumesc, steluta mea norocoasa!*

Lucian Voinea
August 2007

Curriculum Vitae

Stefan-Lucian Voinea was born on May 22nd 1976 in Constanta, Romania. He followed undergraduate studies at Politehnica University in Bucharest, where he obtained in June 2000 his Bachelor and Master degrees, both in Computer Science and Electrical Engineering. In 2001 he moved to The Netherlands where he followed the Software Technology postgraduate studies of the Stan Ackermans Institute in Eindhoven. In September 2003 he obtained his Master of Technological Design title. Since October 2003 he has been a PhD student at Technische Universiteit Eindhoven under the supervision of dr. ir. A.C. Telea, prof. dr. ir. J.J. van Wijk and dr. J.J. Lukkien. His research concerns visualization of software evolution, and has led amongst others to several publications at international conferences and in journals, and to this thesis.