How Developers Drive Software Evolution

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Mauricio Seeberger
January 2006

Supervised by
Prof. Dr. Oscar Nierstrasz
Institut für Informatik und angewandte Mathematik

The address of the author:

Mauricio Seeberger
Wylerringstrasse 17
CH-3014 Bern
Switzerland
mseeberger@gmx.ch
http://www.coffeeoracle.ch

Abstract

As software systems grow, reverse engineering is becoming an increasingly important task. The larger the system grows the more complex it becomes and the more effort must be put in to understand it. Consequently, the knowledge of the developers becomes more and more critical for the process of understanding the system. However, in large systems not all developers know about the entire system. Thus, to make the best use of developer knowledge, we need to know which developer is knowledgeable in which part of it.

This thesis aims to provide a lightweight approach to understand how developers changed the system, when and where they worked and which developer owned which part of the system. To answer them, we define the *Ownership Map* visualization based on the notion of code ownership and measurements. We semantically group files and identify behavioral patterns of the developer's work

Acknowledgments

First of all I want to thank Dr. Tudor Gîrba for his kind supervision. He always supported and motivated me with constructive ideas that helped me to improve my thesis.

I also thank Prof. Dr. Oscar Nierstrasz, head of the SCG, for giving me the opportunity to work in his group.

Thanks also to Prof. Dr. Stéphane Ducasse for many motivating discussions. I very much enjoyed the discussions about SmallTalk and OOP.

Special thanks go to Adrian Kuhn for the time consuming pair programming sessions and great ideas when I could not see the jungle anymore because of too many trees, or vice versa. It was a great experience and benefit to work with him on CodeFoo, too. Thanks also to all the other members of the SCG, especially those that distracted me during their coffee breaks.

Further I thank Prof. Dr. Michele Lanza for he introduced me to the SCG when I was tutor in the ESE lecture.

Moreover, I want to thank all the people beyond university, whose lives were affected by this work. Especially Andreas Polyanszky, he always had an ear for answering my silly questions and motivated me while playing chess when I better had to work on my thesis.

Finally I thank my love Susanne Wenger for being in my life, sustaining and encouraging me in spite of hard times.

Mauricio Seeberger,
January 2006

Table of Contents

Chapter 1

Introduction

As software systems grow, reverse engineering is becoming an increasingly important task. Changing requirements challenge the original design. Parts of the system have to be fixed, extended or replaced to fulfill the new requirements. The larger the system grows the more complex it becomes. The more complex a software system is, the more effort must be put in to understand it.

Even if the original documentation exists, it might not reflect the code anymore. In such situations, the knowledge of the original developers is the best documentation available. To understand the software systems it is crucial to get access to this knowledge. However in large systems not all developers know about the entire system. Thus, to make the best use of developer knowledge, we need to know which developer is knowledgeable in which part of the system.

An important "First Contact" reengineering pattern says "Chat with the maintainers" [Demeyer *et al.*, 2002]. But time is scarce and in the early stages of reverse engineering we need quick results. We do not have time to ask every developer and some of them may even not be available. Which developer should we ask first?

From another perspective, Conway's law [Conway, 1968] states that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." That is, the shape of the organization reflects on the shape of the system. As such, to understand the system, one also has to understand the interaction between the developers and the system [Demeyer *et al.*, 2002].

1.1 Analyzing Versioning Systems

Using versioning systems has become popular for analyzing changes to software source code. As they store the changes that developers make to their software system, they are also the best source for code change analysis. Moreover, because of the wide use of versioning systems in open-source projects, it has become easy to get access to the information needed for code change analysis research.

This thesis aims to understand how the developers drove the evolution of the system. In particular we provide answers to the following questions:

— How many authors developed the system?

— When and where worked the developers?

— What were the behaviors of the developers?

— Which author developed and owned which part of the system?

To find out which developer is the most knowledgeable in a part of the system, a definition of ownership is needed. In our approach, we assume that the developer who worked the most on a file is the most knowledgeable in that file. Based on this assumption we define the owner of a file not being the one that introduced the most lines, but the one that changed the most of them.

We make use of metrics that calculate the proliferation and focus of the author's work. Based on the ownership and these measurements we provide a visualization that helps us to understand how developers interacted with the system. The visualization represents files as lines, and colors[1] these lines according to the ownership over time, see Chapter 4 (p.23). In addition we give a semantic order to the file axis of the visualization what allows us to identify behavioral patterns about the work of the authors.

We implemented our approach in Chronia, a tool built on top of the Moose reengineering environment [Ducasse *et al.*, 2005]. As the Concurrent Versioning System (CVS)[2] is a de facto versioning system, our implementation relies on the CVS model. Additionally CVS can provide a log file that contains only meta information about the changes. In fact this log information is sufficient for building up our approach. Thus, we do not need to check out every file revision in order to extract the information we need. Our aim was to provide a solution that gives fast results, therefore, we tuned our approach to rely only on information from the log file. Retrieving the CVS log is fast and minimizes the network traffic compared to retrieving every file revision from the entire software

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

[2] http://www.nongnu.org/cvs/

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

To show the usefulness of our solution we applied it on several large case studies. In Chapter 5 (p.31) we do extensive experiments and report the findings discussing different facets of the approach.

1.2 Contributions

This thesis provides lightweight approaches for understanding the interaction between the developers and the system. The goal is to have a simple, but powerful mechanism to get quick answers to the questions in Section 1.1 (p.2). To reach this goal, the following contributions were developed in detail [Gîrba *et al.*, 2005a]:

1. To find out which developer is the most knowledgeable in a file, a definition of file ownership is needed. Based on the assumption that the developer who worked the most on a file is the most knowledgeable in that file, we determine the owner of a file as being the developer who changed the most lines of that file.

2. To detect patterns in the evolution of the system, the files need to be ordered in a meaningful way. We define the commit signature of a file and the distance between two files using a modified Hausdorff distance. By clustering the files based on the distance of their commit signature, we give a semantic order to the file axis.

3. Based on the definition of the file ownership and using proliferation and focus metrics we provide a visualization that helps to understand how developers interacted with the system. The visualization called *Ownership Map* represents files as lines, and colors these lines according to the ownership over time.

4. Ordering the files in a semantic way in the *Ownership Map* reveals semantical information about the work of the developers. We identified several developer behavioral patterns and give a name and description to each of them.

5. Our approach is implemented in Chronia [Seeberger *et al.*, 2006], a tool built on top of the Moose reengineering environment. Chronia builds an *Ownership Map* directly out of a CVS repository over the network and allows for interactive exploration of the *Ownership Map* using zooming and contextual menus. It also allows customization of several rendering and ordering preferences. It is implemented in SmallTalk.

1.3 Document Structure

Chapter 2 (p.5) browses the state of the art in code change analysis, particularly regarding the integration of authorship information. First, we show approaches in data processing and modelling. Then the chapter presents visualizations distinguishing between anonymous visualizations and those that process author information. Further we present work on the co-change analysis field and conclude with a discussion of the visualizations and the presented approaches.

In **Chapter 3** (p.15) we define how we measure the code ownership using the information available in versioning systems. We explain the shortcomings of the CVS versioning system and how we solved them. At last, we present the definition of proliferation and focus metrics that characterize the work of the authors.

We use these measurements in **Chapter 4** (p.23) and introduce our *Ownership Map* visualization. We explain the semantical ordering of its axes and explain how to read and interpret it. Using this semantical ordering we can detect behavioral patterns that characterize the work of the authors.

Chapter 5 (p.31) validates the usefulness of our by applying the *Ownership Map* and measurements. We do experiments on several large case studies and report the findings telling the story of the system and of the authors.

Chapter 6 (p.47) shows the details of Chronia, our implementation of the approach. Chronia is an interactive tool written in SmallTalk built on top of the Moose reengineering environment [Ducasse *et al.*, 2005].

Chapter 7 (p.53) concludes our approach with some discussions of open points and limitations of our approach. Finally we list the future work that we would like to invest in our approach.

Chapter 2

Code Change Analysis and Author Information

2.1 Introduction

In the recent years much research effort has been dedicated to understand software evolution, showing the increasing importance of the domain. Code change analysis can focus on a variety of points, *e.g.,* on the changes itself, on the evolution of the system, on the authors doing the changes, *etc.* Most of the research focused on the analysis of system structure evolution, but analyzing the way developers interact with the system has only attracted few research. We call the analysis, that does not take the authors that developed the system into account, to be "anonymous" analysis.

To understand how the system evolved over time, the changes made to the system have to be analyzed. Common data sources are versioning systems. Apart from the actual changes, they also store when and who made the changes – essential information for code change analysis. Surprisingly, most analysis that access versioning system data build upon the what and when information. They treat the changes as anonymous events during the evolution of a software. Some researches use the author name to tag the changes, but do not focus on the author history. They analyze the evolution from the perspective of the system and not from the perspective of the developers.

It is the developers that develop the software system. It is also the developers that have the best knowledge about the system they have built. We can analyze a software system without caring about the developers. But what if there still are some of the original

developers available? Shouldn't we ask them first, before we spend a lot of time to figure out how the system works? Thus, gaining access to the developer's knowledge, in order to faster understand the software system, plays an important role in a reengineering process. And to get access to the developer's knowledge we must first know which author is knowledgeable in which part of the system.

The main challenge of code change analysis is the large amount of data that has to be processed. Usually, the older a system the more changes have to be taken into account. This can lead to problems of scalability, due to time and resource consuming calculations. Clever data preprocessing and modeling can solve most resource problems. Nevertheless, with increasing data, the presentation of the results is not trivial anymore. It has to be presented in a useful way in order to be valuable to software reengineers.

Common techniques to present this type of data are metrics and visualizations. The largest part of visualizations come from anonymous code change analysis. To prevent scalability problems, most visualizations either focus on single system entities, such as files, classes, . . . , or on the evolution of the system and its modules, . . . , *etc.* The latter leads to less detailed information about the change content but results in being a very valuable entry point for system history analysis.

This chapter gives an overview about the state of the art in code change analysis. As we are interested in which developers changed which part of the system, we particularly regard the integration of authorship information in the different approaches we present.

Structure of the Chapter

First, some approaches on data processing and modeling are presented in Section 2.2 (p.6). Then we show the related work done in code change analysis, dividing it into two main parts, "metrics and visualizations" Section 2.3 (p.8) and "co-change analysis" Section 2.4 (p.12). Finally Section 2.5 (p.13) concludes the chapter summarizing and discussing the results.

2.2 Data Processing and Modeling

Most versioning systems store and provide the information in a way not appropriate for evolution analysis. Their main purpose is to store and manage the data and not to analyze it. In order to perform the analysis, the retrieved data has to be (1) preprocessed,

(2) modeled and (3) stored to provide further access. This is an often underestimated effort. This section presents one approach for each one of this tasks.

Software systems accumulate large amounts of data during their evolution. Resource and time limitations make it often impossible to process all available information. The data has to be summarized first. Furthermore some versioning systems even lose information that might be relevant for code change analysis. Sometimes this information can partially be recovered using sophisticated techniques. But again, this algorithms often require intensive calculations.

CVS, for example, does not store the transactions made by the authors. They have to be recovered from the file revisions. Zimmermann and Weißgerber [Zimmermann and Weißgerber, 2004] pointed out this and other problems that appear when preprocessing CVS data. They introduced techniques for recovering transactions (see also Chapter 3 (p.15)) and for mapping files to entities. Unfortunately they did not focus on the developers but focus on preprocessing the data for anonymous analysis.

After preprocessing the versioned data, a model is needed to hold the data in a useful way for further access. While most research use their own ad-hoc model for their explicit task, a generic meta-model would benefit the reuse of evolution analysis results.

Tudor Gîrba presents the meta-model *Hismo* and shows the advantages of modeling history as a first class entity [Gîrba, 2005]. The basic idea is to have a history entity that contains several versions which refer to a snapshot. Figure 2.1 (p.7) shows the relationships between history, version and snapshot. The snapshot is a placeholder for the entity whose evolution is studied *i.e.,* file, package, class, *etc.* A version adds time information to a snapshot. It knows its history and the succeeding and preceding version, if any. At last, the history is an ordered list of versions.

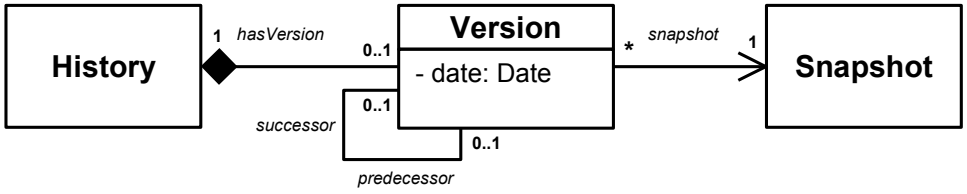


Figure 2.1: Concept of the Hismo meta-model.

A perfect versioning system would eliminate the need of preprocessing the data in order to recover hidden information like the CVS commits, or renamed files, *etc.* It would provide all required information for evolution analysis and present the perfect model. But we can not choose the versioning system that should hold the software's history we

want to analyze. We have to deal with the information that is available from the current versioning system. The solution out of this dilemma can be a wrapper around the actual versioning system that provides a better interface for evolution analysis.

Draheim and Pekacki [Draheim and Pekacki, 2003] present the meta-model of Bloof, an interface to versioning systems designed for analytical access. It's aim is to remove the high barrier of access to software project data. Bloof is written in Java and the interface is realized as Java-API. They showed how to define metrics using their interface. Apart from metrics like "amount of changes per month" and "changed lines of code per day", they also present developer related metrics such as "team collaboration", where they compared the total changes with collaborative changes on a daily basis.

There are much more and sophisticated methods to recover lost information from versioning systems. Data preprocessing and modeling goes far beyond the focus of this thesis. In fact, most researches have their own way of preprocess and model versioned data.

2.3 Metrics and Visualizations

Measurements and visualization have long been used to analyze how software systems evolve. Visualizations usually base on measurements data that they display in any kind. Measurements, however, can also stand of their own and be used to understand the history of softwares [Mockus *et al.*, 1999; Mockus and Votta, 2000; Mockus and Weiss, 2000].

Focussing on author information, code change analysis visualizations can be divided into two groups: anonymous approaches and approaches that process authorship information. The anonymous visualizations just ignore author information *i.e.*, they treat changes as anonymous events in the software evolution. The other group of approaches – in the context of this thesis the more interesting ones – process the available information, related to the authorship of the changes.

2.3.1 Anonymous Code Change Visualizations

The research of code change analysis not processing authorship information is a field of great interest for author oriented evolution analysis. The approaches to detect code change patterns are similar if not the same and are of significant use to understand how the developers changed the system.

A popular way of visualizing code change is to plot the changes as a function of time: one axis representing the time, the other axis showing versioning information *e.g.*, measurements. [Gall *et al.*, 1997]. Such two-dimensional views are clear to read as you have one axis representing the time line of the software project.

Having only one axis free for mapping measurements limits the use of these visualizations. To enhance the information content, we can plot different shapes, *e.g.*, circles, rectangles, *etc.* and mapping additional information on the dimensions and colors of the shapes [Godfrey and Tu, 2000; Capiluppi *et al.*, 2004].

Other types of views, *e.g.*, matrix views, 3D views, bar and pie charts, *etc.*, are also frequently used to visualize and summarize code change measurements. For a discussion of the usage of the different visualization types see [Eick *et al.*, 2002].

Lanza and Ducasse visualize the evolution of classes in the Evolution Matrix [Lanza and Ducasse, 2002]. Each class version is represented using a rectangle. The size of the rectangle is given by different measurements applied on the class version. From the visualization different evolution patterns can be detected such as continuous expansion, growing and shrinking phases *etc.*

Gîrba *et al.* [Gîrba *et al.*, 2005b] introduced the notion of first class history entities like class hierarchy history, inheritance history, history of properties, *etc.* They defined measurements to summarize their evolution and used them to define rules for detecting characteristics of the evolution of class hierarchies like “young”, “old”, “stable”, “balanced”, *etc.*. They developed a visualization called *Hierarchy Evolution Complexity View*, an evolutionary polymetric view [Lanza, 2003] that uses a tree layout to visualizes the histories of classes and of inheritance relationships

Jazayeri analyzes the stability of the architecture [Jazayeri, 2002] by using colors to depict the changes. From the visualization he concluded that old parts tend to stabilize over time.

Rysselberghe and Demeyer use a scatter plot visualization of the changes to provide an overview of the evolution of systems and to detect patterns of change [Van Rysselberghe and Demeyer, 2004]. They order the files alphabetically, what may distort the recognition of change patterns. An author might change files that are not near each other in alphabetical order, but still correlate with each other.

2.3.2 Visualizing Authorship Information

In the previous Section 2.3.1 (p.8) we got an overview about anonymous analysis. The other part of visualizations in code change analysis are the visualizations that make use of the authorship information available in versioning systems.

Capiluppi used two-dimensional visualizations and plots different measurements extracted out of open source projects to analyze their evolution [Capiluppi, 2003]. He put the number of authors into correlation with the size of the projects and with the number of modules.

Chuah and Eick proposed three visualizations for comparing and correlating different evolution information like the number of lines added, the errors recorded between versions *etc.* [Chuah and Eick, 1998]. Developer information is considered in one property only, in the "number of people working" in the project at the same time. They analyzed how these properties changed over time revealing trends or anomalies of the code evolution.

Xiaomin Wu *et al.* visualized [Wu *et al.*, 2004b] the change log information to provide an overview of the active places in the system as well as of the authors activity. They display measurements like the number of times an author changed a file, or the date of the last commitment. In their nested graph visualization of the system the authorship is displayed as tooltip and/or in the color of the entities.

Ball and Eick [Ball and Eick, 1996] developed visualizations where they reduce a line of source code to a single row of pixels (keeping the indentation). They mapped the color of the lines to different metrics, such as the author, the number of changes or even the number of execution times of the code line gained from runtime profiling.

Jingwei Wu *et al.* use the spectrograph metaphor to visualize how changes occur in software systems [Wu *et al.*, 2004a]. They used colors that fade out to denote the age of changes on different parts of the systems. A second spectrographic visualization plots the changes of each developer on the time axis showing the number of subsystems the changes affected.

Eick *et al.* proposed multiple visualizations to show changes using colors to denote the developer and width to map the module size [Eick *et al.*, 2002]. They also discuss different types of views (matrix, 3D, bar and pie charts, *etc.*) about their usability and use this views to show change metrics. These change metrics include change count by developer, severity and status of the change.

An interesting visualization was used to visualize how authors change a wiki page [Viégas *et al.*, 2004]. Each sentence of an article is visualized with a line in the color of the author. Using the edit history of a wikipedia article, they detect edit patterns like "vandalism and repair", "collaboration", "negotiation", *etc.* using some metrics and their visualization called *history flow*.

Voinea *et al.* implemented a tool called CVSscan. They analyze files out of CVS repositories and show all versions of the text lines [Voinea *et al.*, 2005]. They use colors to encode attributes like "author", "construct", and "line status" on each of these line

versions.

2.3.3 Discussion on Visualizations

An important problem of visualizations is their scalability. While for a small featured project the visualization fits well, it could turn out to be absolutely useless for a very large project with thousands of entities (*e.g.,* files, classes, methods, *etc.*). This is particularly a problem for views where each entity is visualized separately. The user can not make use of thousand windows, *e.g.,* one for each class like the visualizations of [Ball and Eick, 1996]. This problem gets even worse when we think in terms of code evolution. Each entity's history consists of several versions that have to be visualized in an appropriate way.

The solution can be to summarize or condense the information to be displayed. Instead of visualizing each entity separately, one could have a single view showing all entities and mapping the information values to the dimension, form, color of *e.g.,* a rectangle.

Another way of avoiding scalability problems is to make the visualizations interactive. For example, in providing a zooming mechanism the user can scale in if he is interested in a specific part *e.g.,* time period, of the displayed data. Tooltips or context menus for opening new views are also powerful solutions. We have seen among the presented research both, static and interactive views.

But relying too much on interaction is often less useful than keeping the visualization simpler but static. On printed media the benefit of interactivity gets lost.

Most of the interactive views use the interactivity to control the view and do not require it in order to read and interpret the visualization. This is not anymore true for visualizations that display more than two dimensions *e.g.,* 3D visualizations. Here the interactivity is *needed* to prevent occlusion. Because of the perspective view, shapes at the front are rendered larger than the ones toward the rear and therefore tend to be interpreted as more important. There may be even shapes that obscure smaller ones what leads to information loss. As a conclusion, we do not favor 3D visualizations, though they may look nice they offer more problems than benefits.

Some visualizations are intended to look for patterns, others help to understand the software evolution and others are generated for documenting purposes. The visualizations make it up to the user to interpret them, though more sophisticated algorithms may mark or emphasize areas of interest.

2.4 Co-Change Analysis

Another related domain is the analysis of the co-change history, *i.e.,* the analysis of changes occurring more or less at the same time. In this domain too, the changes can be analyzed as anonymous events or they can be combined with developer information.

Gall *et al.* aimed to detect logical coupling between parts of the system [Gall *et al.*, 1998] by identifying the parts of the system which change together. They used this information to define a coupling measurement based on the fact that the more times two modules were changed at the same time, the more they were coupled.

Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graphs [Hassan and Holt, 2004]. They present different heuristics for predicting change propagation. These include a developer based heuristic, which assumes that change propagates to other entities changed recently or frequently by the same developer.

Zimmerman *et al.* aimed to provide a mechanism to warn developers about the correlation of changes between functions. The authors placed their analysis at the level of entities in the meta-model (*e.g.,* methods) [Zimmermann *et al.*, 2005]. The same authors defined a measurement of coupling based on co-changes [Zimmermann *et al.*, 2003]. Though their tool is intended to be used by developers, they do not evaluate any developer specific information in the changes. Every change is weighted the same way: by its age. But changes of a developer, that owns the subsystem, may be more important as changes from a bugfixing author.

Not only information from versioning systems are of interest. In large projects the developers do not only commit code changes, but also write documentation, mails, bugreports, *etc.* This information can also be used to detect change patterns, though it multiplies the data processing and modeling effort.

Čubranić and Murphy bridged information from several sources to form what they call a "group memory" [Čubranić and Murphy, 2003]. Čubranić details the meta-model to show how they combined CVS repositories, mails, bug reports and documentation [Čubranić, 2004]. They process all available information and provide an interface to this group memory called Hipikat. It recommends to a developer artifacts selected from the project memory that may be relevant to the task being performed.

This section summarized how most actual co-change research uses correlating changes to detect and to document change propagation. The author's information are only considered to tag the changes but do not play a deeper role. The co-changes can also be

used to detect patterns in the behaviors of the authors in similar way like the analysis of the wikipages [Viégas *et al.*, 2004].

2.5 Conclusion

This chapter presented the research work of code change analysis regarding the evaluation and integration of author information.

Having wide availability of versioning systems, actual code change analysis research is beginning to take the authorship information into account. The approaches differ in the level of evaluating this information. Some researches make system structure evolution analysis and tag the changes with the developer name, only. Others already make sophisticated use of the available information like the *history flow* visualization of wikipedia articles [Viégas *et al.*, 2004].

Furthermore, what most research has in common is the perspective of interpreting the extracted author information. The focus lies on the analysis of the system history evolution. The author information is then provided as additional information to understand the system structure. This is of course a necessary task to understand the system. But in a reengineering process before you try to understand the system in detail, you want to get an overview about the history of the system and the developers [Demeyer *et al.*, 2002]. If you are lucky and there are still some original developers in the project, you may even save yourself a lot of time asking the right questions to the right developer. But almost no research concentrates on understanding the history of the developers and how they worked.

This thesis aims to provide a lightweight approach to understand how developers changed the system, when and where they worked and which developer owned which part of the system. To answer these questions, we combine both, system and developer history information. Focussing on the developers, we will analyze software projects using self developed and already available measurements. We introduce the *Ownership Map* visualization and show how to interpret it to find the requested answers.

Chapter 3

Measuring How Developers Change the Software Systems

3.1 Introduction

This chapter introduces measurements that are needed for our approach. The main target is the definition of code ownership of a piece of software code *e.g.*, file. We assume that the developer who worked the most on a file is the most knowledgeable in that file. Based on this assumption we define the owner of a file not being the one that introduced the most lines, but the one that changed the most of them.

Moreover, we present the definition of proliferation and focus metrics. These measurements allow us to characterize the work of the author and help to underline statements made from interpreting the *Ownership Map* visualization we will present in Chapter 4 (p.23).

Structure of the Chapter

First we show how to retrieve the required information from versioning systems in Section 3.2 (p.16). Section 3.3 (p.17) shows how to measure the initial size of a file, a needed measurement for the calculation of the change size in Section 3.4 (p.18) and the definition of code ownership in Section 3.5 (p.19). Section 3.6 (p.20) explains further metrics we use in our case studies. Finally, Section 3.7 (p.21) summarizes this chapter.

3.2 Versioning Systems

To analyze the history of software evolution and the way authors changed the system, we need the source code of different points in the software lifetime. The more versions we can analyze, the more accurate are the interpretations of how the developers change the software.

Using versioning systems has become popular for managing changes to software source code. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing those changes the next day. But the usefulness of version control software extends far beyond the bounds of the software development world. As those versioning systems store when, who and what was changed, they are also the best source for code change analysis.

It is crucial to have access to this information if we want to analyze the history of a software. Thus to have good case studies with several authors, we have chosen to focus on the Concurrent Versions System (CVS), as it is the most common versioning system and in wide use in Open-Source projects. Most CVS repositories of well-known open-source projects are accessible anonymously, so that everyone can check the files out, without having to register and create accounts. Moreover, the simple architecture of CVS makes it easier to integrate a client with already available analysis infrastructure.

3.2.1 CVS in Detail

The most straightforward approach to analyze the software's history is to check out all file versions ever committed to the versioning repository and to compute the code ownership from diff information [MacKenzie *et al.*, 2003] between all subsequent revisions f_{n-1} and f_n. From an implementation point of view this implies the transfer of large amounts of data over the network, and long computations, but will probably work with all sort of versioning systems. A second problem is the copyright of the source code most industry software systems have. It is not easy to get access to the source code then, though it would be interesting to analyze it.

We aim to provide for an approach that can deal with large projects with long history, and that can provide the results fast. Thus we tuned our approach to work with other information CVS can provide. Nevertheless, our approach is not bound to CVS. The calculation of the measurements presented in this chapter would be different, depending on the versioning system chosen, but the approaches and its findings will keep the same.

We compute the ownership of the code based only on the CVS log information. In

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

Figure 3.1: Sample CVS log snippet.

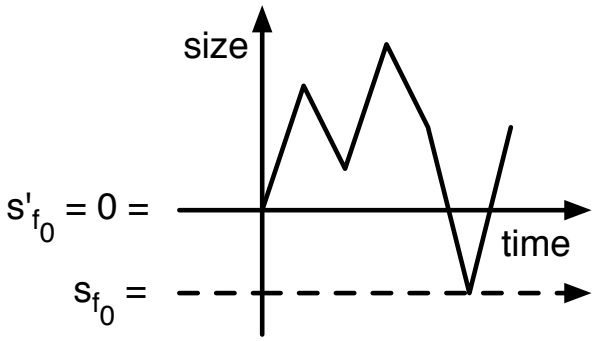
Figure 3.2.1 (p.17) we present a snippet from a CVS log. The log lists for each version f_n of a file – termed revision in CVS – the time t_{f_n} of its commit, the name of its author α_{f_n}, some state information and finally the number of added and removed lines as deltas a_{f_n} and r_{f_n}. We use these numbers to recover both the file size s_{f_n} and the code ownership $own^{\alpha}_{f_n}$, both presented in the next two sections.

3.3 Measuring File Size

Let s_{f_n} be the size of revision f_n, measured in number of lines. The number of lines is not given in the CVS log, but can be computed from the deltas a_{f_n} and r_{f_n} of added and removed lines. Even though the CVS log does not give the initial size s_{f_0}, we can give an estimate based on the fact that one cannot remove more lines from a file than were ever contained. We define s_{f_n} as in Figure 3.2 (p.18): we first calculate the sizes starting with an initial size of 0, and then in a second pass adjust the values with the lowest value encountered in the first pass.

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

Moreover it lowers the impact on the ownership of imported files. In CVS files can either be added or imported to the repository. The import functionality of CVS lets an



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

Figure 3.2: The computation of the initial file size.

author recursively add directories of files. But the author who adds all these files is not necessary the author of the files. Hence, if the developer imports files containing a lot of lines he would be the owner of that file for its whole lifetime, even if he does not work on that file. With the approach of estimating the initial size, the importing developer obtains only as much ownership as lines were removed in the file, allowing other authors to take over the files.

3.4 Measuring Change Size

To have a value of how large a change in a file was, we look at the added and removed lines. Here again, a_{f_n} are the added lines and r_{f_n} the removed lines. s_{f_n} is the line size as calculated in Section 3.3 (p.17) for the file revision f_n. The size of the change c_{f_n} that lead to the file revision f_n is calculated the following way:

$$c_{f_0} := 0$$
$$c_{f_n} := \begin{cases} 0, & s_{f_{n-1}} + s_{f_n} \leqslant 1 \\ 1, & a_{f_n} + r_{f_n} > s_{f_{n-1}} + s_{f_n} \\ \frac{a_{f_n} + r_{f_n}}{s_{f_{n-1}} + s_{f_n}}, & else \end{cases}$$

To calculate the size of a change we not only look how much was added, but also how much was removed. Dividing it by the file size we get a value that is relative to the size of the file. Thus, the value of c_{f_n} reflects the proportion of the file that got changed.

Because of some abnormalities of CVS that make it possible to add lines while deleting the file, we force the change to be equal to 1 to prevent too high values. For example if we have a file with 2 lines, and we add 100 lines and delete the file (remove 102 lines), then we would get a change size of $\frac{100+102}{0+2}$ that is equal to 101.

3.5 Measuring Code Ownership

A straight forward approach to define the owner of a file would be to track which author changed which line of the file. The one who most recently changed the line would be the owner of that line and the one who owns the most lines in a file is then the owner of that file. For CVS this would mean to check out the content of all previous versions of the file. Using a diff algorithm we could find out to whom the removed lines actually belonged. But this requires vast amounts of network traffic and time consuming calculations and the advantages of only processing information from CVS log would be lost.

If we want to provide fast results we need an approximation of the real code ownership only based on information available in the CVS log. Given the file size s_{f_n}, the author α_{f_n} that committed the change and a_{f_n} the number of lines he added, we approximate the ownership as:

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

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

In this definition we assume that the removed lines r_{f_n} are evenly distributed over the ownership of the preceding owners of f_{n-1}. Let's explain this with an example: Assume that author A and author B each add 10 lines to a file so that $own^{\mathsf{A}}_{f_1} = own^{\mathsf{B}}_{f_1} = \frac{1}{2}$. Then author B removes his ten lines and replaces them by 1 single line. Now $own^{\mathsf{A}}_{f_2} = \frac{5}{11}$ and $own^{\mathsf{B}}_{f_2} = \frac{6}{11}$, where in truth author A still owns 10 lines and B only 1. Our definition does not take into account which author owned the removed lines. The removed ten lines are evenly distributed among both authors. Author A owns now only five lines and author B owns five plus the one line added, making him to be the new owner of that file.

At first glance this seems to be a big mistake in the calculation of the ownership. Focusing only on the lines of code, it is surely not the best approach. But we are not interested in which author introduced exactly which percentage of the lines of the file. We are interested in which author worked the most on the file. This author should then

also be the owner of that file, even if he does not have written the largest part of it. Our definition of code ownership is a tradeoff between the author who owns the most lines and the author who changed the most of the file. This means that $own^{\alpha}_{f_n}$ is an approximation of the percentage of lines owned by author α in revision f_n.

3.6 Author Proliferation and Focus

This section presents measurements that we will use in the case studies in Chapter 5 (p.31). These metrics characterize the work of the author as a value representing whether he worked over the whole system or only on specific parts. The idea of these metrics is to support and fortify statements made by interpreting the *Ownership Map*, Chapter 4 (p.23). Though they are not not necessary to build and understand the visualization, they are useful for deciding whether an unclear statement is correct or not.

Suppose we have a developer who did two thousand commits in a project with three thousand files. We do not know whether this author worked only on the same file or has modified more than half of the system. The proliferation metric is a solution that gives an indication of how much the work of the author was distributed over the system.

Let's now suppose that this author modified a third of the system; we still do not know how localized his work was. Did he only work on one of the three main layers of the software? Or did he modify ever third file across the system? The first would make him probably to be an expert of that subsystem layer. The latter would identify him as an allrounder who knows large parts of the system but none very deeply. Here we can apply the focus metric.

3.6.1 Proliferation

The proliferation of a set of files f over a set of sets of files g (group of files) is defined as:

$$\sigma(f, g) := \sum_{g_i} \frac{|f \cap g_i|}{|g_i|}$$

$\sigma(f, g)$ is an indication of how much the set of files f are distributed in the groups g. The maximum value the proliferation can take is equal to the number of groups.

To apply this measurement we need a set of files f and a set of file groups g. A straightforward grouping for g would be to take the top level directories of the project's code base. Taking all available directories of the code base is also a solution. For f we can

take the files an author has modified (touched) in a certain time period or the files that he owns, *etc.* to make statements about the work of the author.

For example, let's say we have 2 directories and a proliferation value of 1 for the files that the author has touched during all periods of system development. We know now that this author has modified half of the directories. This does not mean that he has modified half of the files. It can be that the first filegroup contains 10 and the second only 2 files. If the author has modified only the two files of the second directory, we already get a proliferation of 1. But he could also have modified 5 files in the first directory and 1 file in the second. He would have the same proliferation value.

Thus it is important to have a grouping with equal amount of files in order to make useful statements. The smaller a group the more weight the files have.

3.6.2 Focus

The focus of some files f in some groups g (group of files) is defined as:

$$\tau(f, g) := \sum_{g_i} \frac{|f \cap g_i|}{|g_i|} \frac{|f \cap g_i|}{|f|}$$

The focus tells us how localized the files are in the groups. We weight the proliferation with the percentage of the total files in this group. A low value means that the files are well distributed over the groups. A high value tell us that the files lie all very bounded in the same group(s).

Using the same example as above in the proliferation subsection, we can now distinguish author A who worked only on the 2 files and author B who worked half-half in both directories. The first author would get a focus of 1, which actually is the highest value τ can take. The second author only has a focus value of $\frac{1}{2}$. Now we can say that author A worked more localized on the groups of files than the other one.

3.7 Summary

In this chapter we presented a definition of code ownership and introduced metrics that evaluate the characteristic of developers' work. We know now how to retrieve versioning data from a CVS log and how to process it in order to have the information needed. Then we used this information to present the calculation of the initial file size and defined the

code ownership. Finally we introduced the definition of two metrics that characterize the work of authors.

Though our approach is based on CVS log information, all definitions are independent of CVS. Thus once you can provide the author name, date, added/removed lines, and the initial file size, our approach can be used to analyze the history of the project.

The next chapter uses the metrics and definitions to present the *Ownership Map* visualization.

Chapter 4

The Ownership Map View

4.1 Introduction

To provide a useful approach for analyzing software systems it is not enough to only present numbers and values gained from applying measurements on the software project's history. The human reverse engineer needs intuitive but compact results, so that he does not lose the overview of the whole history and can focus on selective areas of interest. One of the most common techniques to achieve this goal is visualization.

This chapter presents our approach to visualize the history of software systems. As we focus on the authors we are interested in displaying author related attributes. We call our visualization *Ownership Map* because its main focus is to display the file ownership of the developers. In order to render our visualization, we depend on the definitions explained in Chapter 3 (p.15).

Structure of the Chapter

Section 4.2 (p.24) introduces the *Ownership Map* visualization. In Section 4.3 (p.25) we explain how we order the axis of the view in order to detect behavioral patterns that are presented in Section 4.5 (p.28). Finally, we summarize this chapter in Section 4.5 (p.28).

4.2 *Ownership Map* Visualization

We introduce the *Ownership Map* visualization as in Figure 4.1 (p.24). The visualization is similar to the Evolution Matrix [Lanza and Ducasse, 2002]: each line represents a history of a file, and each circle on a line represents a change to that file.

The color of the circle denotes the author who made the change. The size of the circle reflects the proportion of the file that got changed *i.e.,* the larger the change, the larger the circle. To calculate the size of a change we take the added and removed lines into account. See Section 3.4 (p.18) for the definition of the exact calculation of the change size.

The color of the line between the circles denotes the author who owns most of the file, see Section 3.5 (p.19). A missing line means that the file is deleted or not yet existent.

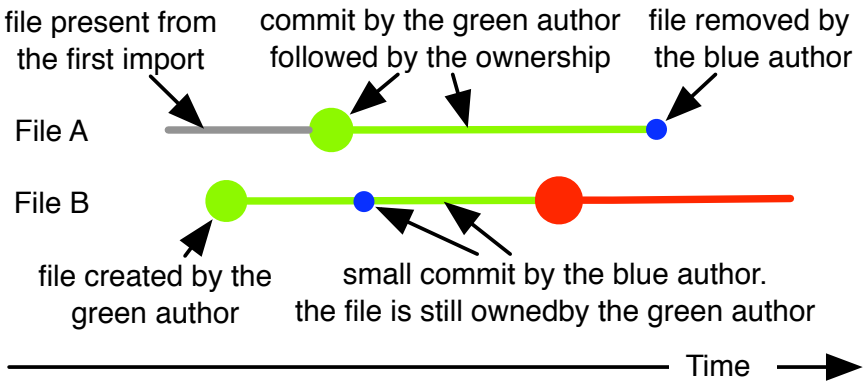


Figure 4.1: Example of ownership visualization of two files.

In the example from Figure 4.1 (p.24), each line represents the lifetime of a file; each circle represents a change. File A appears gray in the first part as it originates from the initial import. Later the green author significantly changed the file, and he became the owner of the file. In the end, the blue author deleted the file. File B was created by the green author. Afterwards, the blue author changed the file, but still the green author owns the larger part, so the line remains green. At some point, the red author committed a large change and took over the ownership. The file was not deleted.

Bertin [Bertin, 1974] assessed that one of the good practices in information visualization is to offer to the viewer visualizations that can be grasped at one glance. The colors used in our visualizations follow visual guidelines suggested by Bertin, Tufte [Tufte, 1990], and Ware [Ware, 2000] – *e.g.,* we take into account that the human brain is not capable

of processing more than a dozen distinct colors.

In a large system, we can have hundreds of developers. Because the human eye is not capable of distinguishing that many colors, we order the authors according to the number of commits and only display the first 18 authors using distinct colors; the remaining authors are represented in gray. Furthermore, we also represent with gray the files that came into the CVS repository with the initial import, because these files are usually sources from another project with unknown authors and are, thus, not necessarily created by the author who performed the import. In short, a gray line represents either an unknown owner, or an unimportant one.

4.3 Axis Ordering

We have seen in Section 4.2 (p.24) how to read the history of single files in the *Ownership Map*. If we plot all file histories of a project one below the other, we can look for patterns. But just rendering them without any order would not allow a meaningful interpretation about the work of the authors. A well defined ordering of the axes is needed before we can give semantic meanings to the patterns.

Contrary to similar approaches [Van Rysselberghe and Demeyer, 2004], we give a semantic order to the file axis (*i.e.,* we do not rely on the names of the files) by clustering the files based on their history of changes: files committed in the same period are related to each other [Gall *et al.*, 1998].

This section shows how we order the axes before we present the behavioral patterns we identified in the *Ownership Map* using these ordering in Section 4.4 (p.27).

4.3.1 Ordering the Files Axis

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

Let H_f be the commit signature of a file, a set with all timestamps t_{f_n} of each of its revisions f_n. Based on this the distance between two commit signatures H_a and H_b can be defined as the modified Hausdorff distance[1] $\delta(H_a, H_b)$:

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

With this metric we can order the files according to the result of a hierarchical clustering algorithm [Jain *et al.*, 1999]. From this algorithm a dendrogram can be built: this is a hierarchical tree with clusters as its nodes and the files as its leaves. Traversing this tree and collecting its leaves yields an ordering that places files with similar histories near each other and files with dissimilar histories far apart of each other.

The files axes of the *Ownership Map* views shown in this thesis are ordered with *average linkage* clustering and *larger-clusters-first* tree traversal. Nevertheless, our tool Chronia allows customization of the ordering. For a short discussion about alternative file clustering read Section 7.2 (p.56)

4.3.2 Ordering the Time Axis

Subsequent file revisions committed by the same author are grouped together to form a transaction of changes *i.e.*, a commit. We use a single linkage clustering with a threshold of 180 seconds to obtain these groups. This solution is similar to the sliding time window approach of Zimmerman *et al.* when they analyzed co-changes in the system [Zimmermann *et al.*, 2005]. The difference is that in our approach the revisions in a commit do not have to have the same log comment, thus any quick subsequent revisions by the same author are grouped into one commit.

A commit in the visualization is ore more circles aligned on the same vertical line. They can be positioned in different ways. A straight forward approach is to put them each after another. Another possibility is to position them linearly, *i.e.*, the distance between the commits is proportional to their difference in time. For the standard *Ownership Map* we prefer the first solution with additional vertical lines marking constant time intervals, *e.g.*, 3 months. The shorter these intervals, the less commits occurred in this period.

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

4.4 Behavioral Patterns

Using the semantical ordering of the files as presented in Section 4.3 (p.25) the *Ownership Map* reveals semantical information about the work of the developers. Figure 4.2 (p.27) shows a part of the *Ownership Map* of the Outsight case study (for more details see Section 5.2 (p.32)). In this view we can identify several different behavioral patterns of the developers.

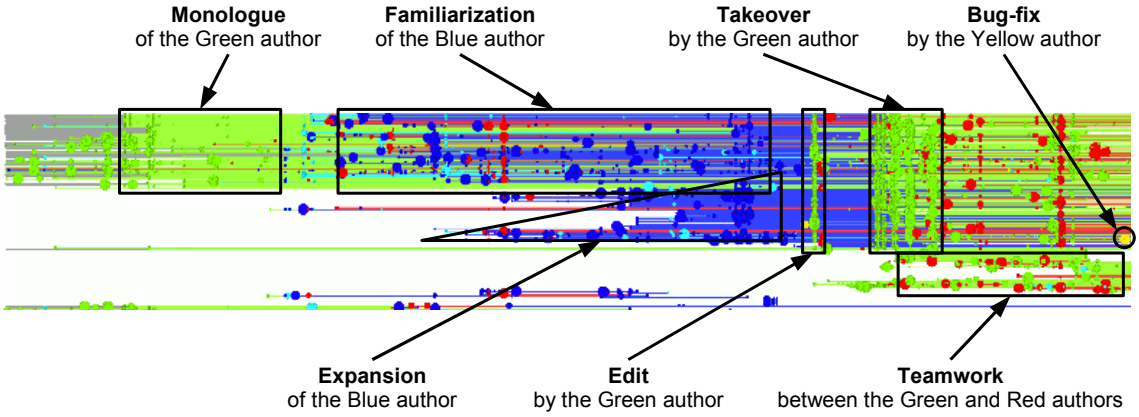


Figure 4.2: Example of the Ownership Map view. The view reveals different patterns: Monologue, Familiarization, Edit, Takeover, Teamwork, Bug-fix.

Applying our approach on several case studies we identified a couple of behavioral patterns. Following we present each of them with a name and a explain what the pattern means and how it looks like on the *Ownership Map*:

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

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

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

Silence. Silence denotes an uneventful period with nearly no changes at all. It is visible on an *Ownership Map* as a rectangle with constant line colors and none or just few change circles.

Takeover. Takeover denotes a behavior where a developer takes over a large amount of code in a short amount of time – *i.e.,* the developer seizes ownership of a subsystem in a few commits. It is visible on an *Ownership Map* as a vertical stripe of single color circles together with an ensuing change of the lines to that color. A Takeover is commonly followed by subsequent changes done by the same author. If a Takeover marks a transition from activity to Silence we classify it as an *Epilogue*.

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

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

Cleaning. Cleaning is the opposite of expansion as it denotes an author who removes a part of the system. We do not see this behavior in the example of Figure 4.2 (p.27).

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

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

4.5 Summary

In this chapter we presented our approach visualizing the history of a software system. We learned how to read the *Ownership Map* and know that it is important to order the axes in a meaningful way in order to detect patterns. Section 4.4 (p.27) presented

a couple of behavioral patterns we identified in our case studies and showed how to interpret them.

With the *Ownership Map* visualization at hand we are able to tell the story of the software project and of its developers as we show in the following Chapter 5 (p.31).

Chapter 5

Case-Studies

5.1 Introduction

This chapter presents the results we found by applying our approach to several case studies. We have chosen several large open source projects and a smaller one, that was developed under closed-source. We start with the smaller project named Outsight. Then we present JBoss, a well-known open-source project. We report detailed findings of the system and the developers of both case studies, and we give an overall impression on three other well-known open-source projects.

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

JBoss. JBoss is an open source Java Application Server written in Java. The CVS repository goes back five years beginning in April 2000 and contains about 2700 files summing up over 25 thousand revisions. The system has been written by 133 authors. 14 of them introduced more than 80% of the code.

Open-source Case Studies. We chose Ant, Tomcat and JEdit to illustrate different fingerprints systems can have on an *Ownership Map*. Ant has about 4500 files, Tomcat about 1250 files, and JEdit about 500 files. The CVS repository of each project goes back several years.

All calculations and visualizations for the case studies were performed on a standard performant computer, *i.e.,* Pentium_m "Banias" 1.4 GHz running on Linux with 512

Mbyte of RAM. The calculations took less than 10 minutes for each case study.

Structure of the Chapter

We present the case studies in the order they were presented in the introduction of this Chapter, Section 5.1 (p.31). To begin we show detailed findings of developer behavior of the Outsight case study in Section 5.2 (p.32). In Section 5.3 (p.36) we go a step further and use metrics and show how they affect the *Ownership Map* of JBoss. Finally Section 5.4 (p.44) lists three other open-source systems and gives a short description of them.

5.2 Outsight

Outsight is a commercial web application written in Java and JSP. It is sort of a web-based job platform, where users can commit their knowledge targeting a solicitation. The CVS repository goes back three years and spans across two development iterations separated by half a year of maintenance. The system has been written by four developers and has about 500 Java classes and about the same amount of JSP pages.

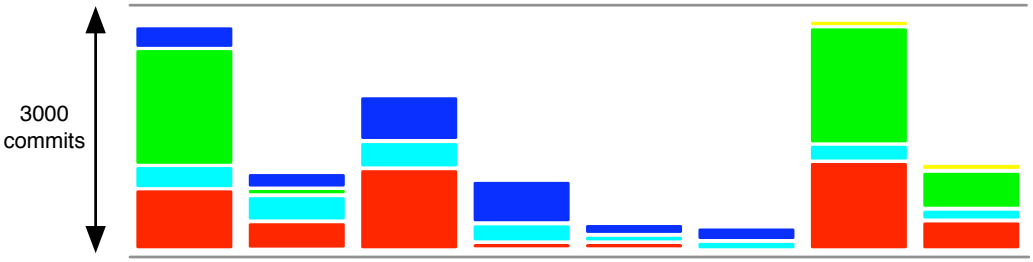
5.2.1 The Story of the System

The first step to acquire an overview of a system is to build a histogram of the team to get an impression about the fluctuations of the team members over time. Figure 5.1 (p.33) shows that a team of four developers is working on the system. There is also a fifth author contributing changes in the last two periods only.

Figure 5.2 (p.34) shows the *Ownership Map* of our case study. The upper half are Java files, the bottom half are JSP pages. The files of both modules are ordered according to the similarity of their commit signature. For the sake of readability we use S1 as a shorthand for the Java files part of the system, and S2 as a shorthand for the JSP files part. The time axis is cut into eight periods P1 to P8, each covering three months. The paragraphs below discuss each period in detail, and show how to read and interpret the *Ownership Map*.

The shorthands in parenthesis denote the labels R1 to R15 as given on Figure 5.2 (p.34).

Period 1. In this period four developers are working on the system. Their collaboration maps the separation of S1 and S2: while Green is working by himself on S2 (R5), the others are collaborating on S1. This is a good example of Monologue versus



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

Figure 5.1: Number of commits per team member in periods of three months.

Dialogue. A closer look at S1 reveals two hotspots of Teamwork between Red and Cyan (R1,R3), as well as large mutations of the file structure. In the top part multiple Cleanings happen (R2), often accompanied by Expansions in the lower part.

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

Period 3. This period is dominated by a big growth of the system, the number of files doubles as large Expansions happen in both S1 and S2. The histogram in Figure 5.1 (p.33) identifies Red as the main contributor. The Expansion of S1 evolves in sudden steps (R9), and as their file base grows the Teamwork between Red and Cyan becomes less tight. In contradiction the Expansion of S2 evolves in small steps (R8), as Blue continues familiarizing himself with S2 and slowly but steady takes over ownership of most files in this subsystem (R6). Also an Edit of Red in S2 can be identified (R7).

Period 4. Activity moves down from S1 to S2, leaving S1 in a Silence only broken by selective changes. Table 5.1 (p.33) shows that Red left the team, which consists now of Cyan and Green only. Cyan acts as an allrounder providing changes to both S1 and S2, and Blue is further working on S2. The work of Blue culminates in an Epilogue marking the end of this period (R8). He has now completely taken over

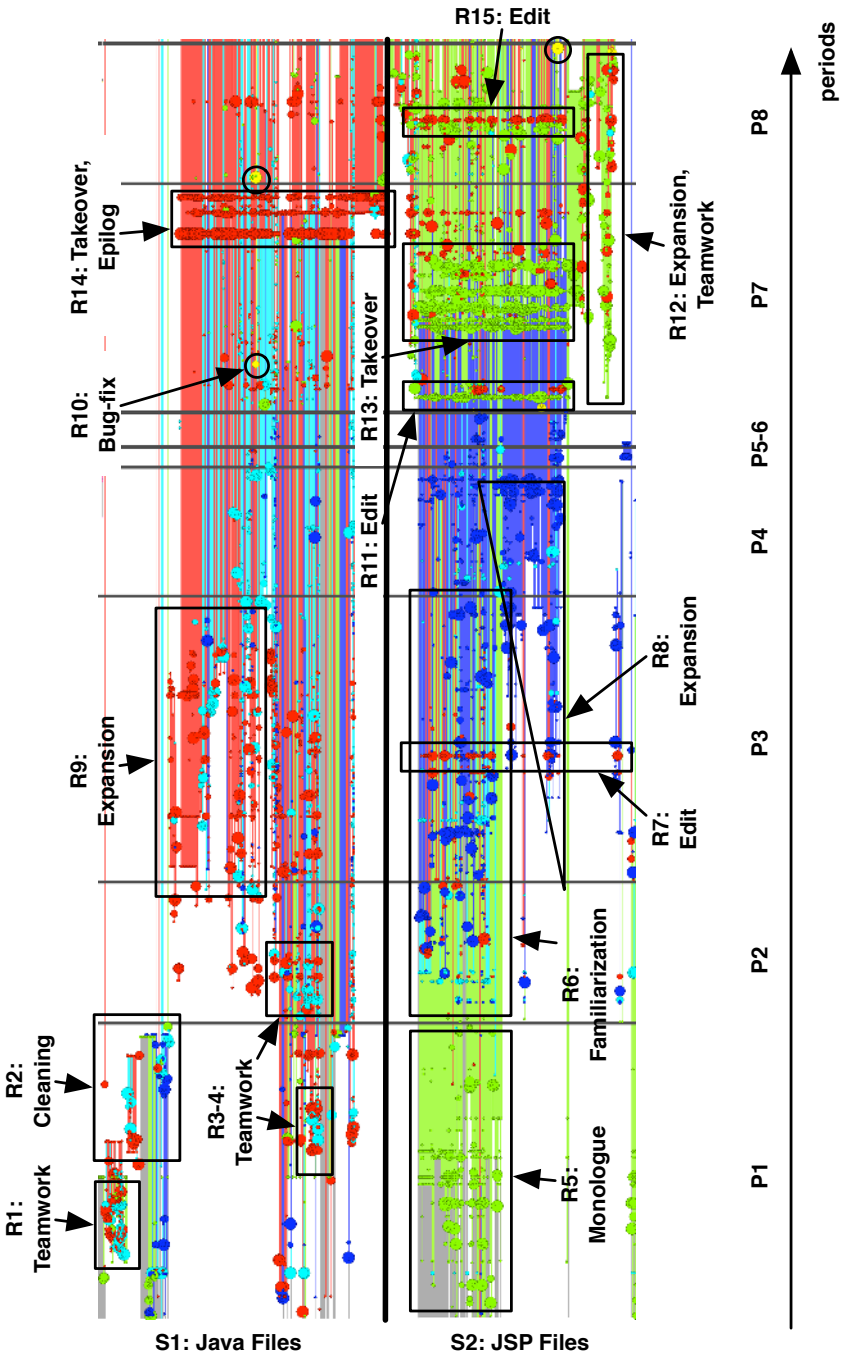


Figure 5.2: The *Ownership Map* of the Outsight case study.

ownership of S2, while the ownership of subsystem S1 is shared between Red and Cyan.

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

Period 7. After two periods of maintenance the team resumes work on the system. In Table 5.1 (p.33) we see how the composition of the team changed: Blue leaves and Green comes back. Green restarts with an Edit in S2 (R11), later followed by a quick sequence of Takeovers (R13) and thus claiming back the ownership over his former code. Simultaneous he starts expanding S2 in Teamwork with Red (R12).

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

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

5.2.2 The Story of the Authors

To focus on the authors we give a description of each author's behavior, and in what part of the system he is knowledgeable.

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

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

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

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

Yellow author. Yellow is a pure Bug-fix provider. But as he only made some few commits during the last two periods of the project, he might not have good knowledge about the system.

5.3 JBoss

JBoss[1] is an open source Java Application Server, a J2EE certified platform for developing and deploying enterprise Java applications, written in Java. The CVS repository goes back five years beginning in April 2000 and contains about 2700 files summing up over 25 thousand revisions. The system has been written by 133 authors. 14 of them have implemented over 80% of the lines, which classifies them as core developers.

Figure 5.3 (p.37) shows the *Ownership Map* of the JBoss case study. Only non binary files are visualized, most of them are Java source code files. The system is split into 3 modules: S1 contains the files that implement the support for Enterprise Java Beans (EJB), S2 the core files of JBoss (those that are in the main-tree) and S3 represents the rest of the files. The files of each module are ordered according to the similarity of their commit signature. The time axis is divided into 11 periods P1 to P11. Each of them spans exactly half a year.

The next two sections discuss the *Ownership Map* in detail. First, in Section 5.3.1 (p.38), we look at the system in general and in Section 5.3.2 (p.38) we tell the story of some of the core developers.

[1] http://www.jboss.org

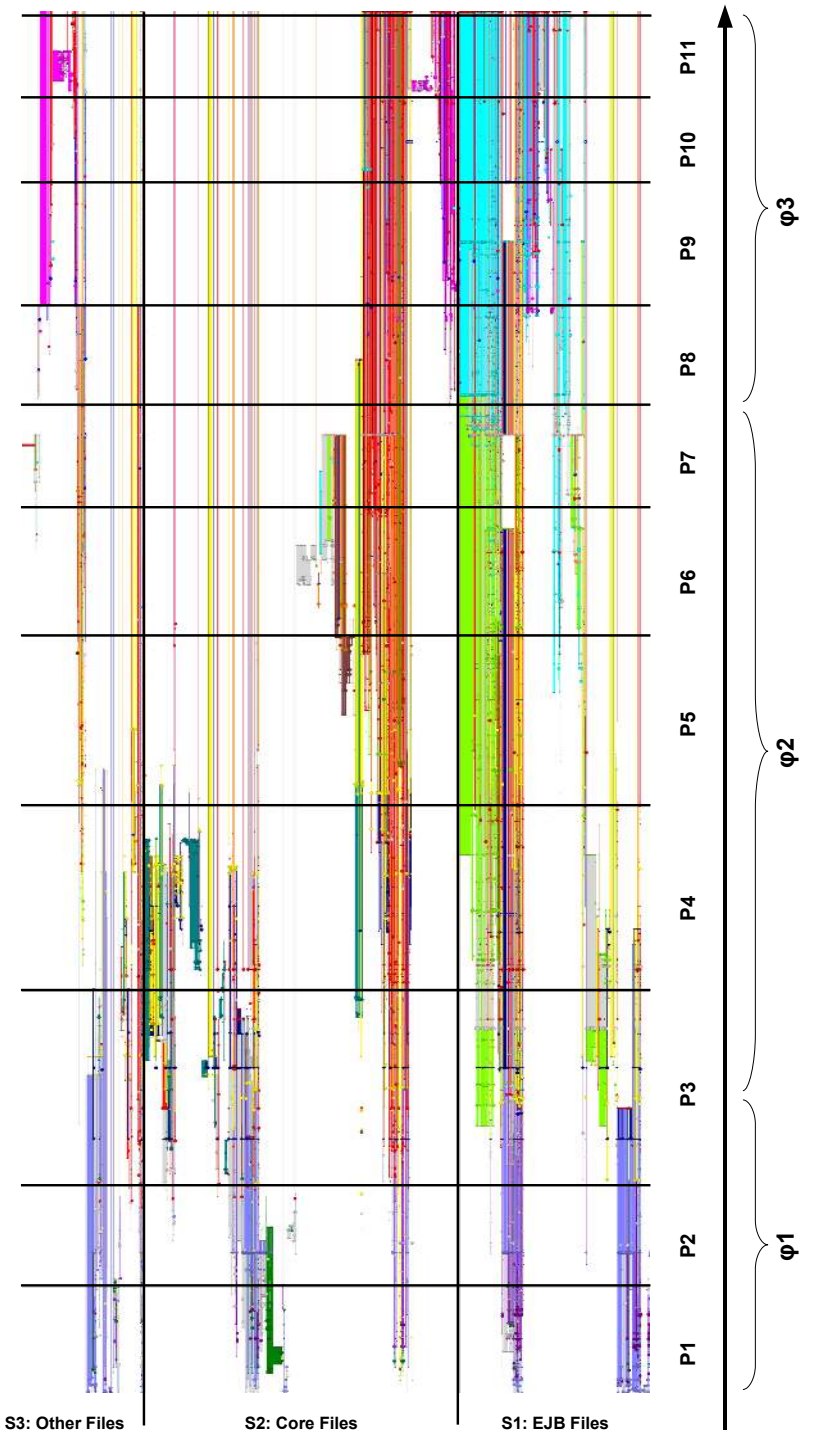


Figure 5.3: The Ownership Map of JBoss.

5.3.1 The Story of the System

The first thing that catches our attention are the long unicolored blocks that change from one color to another. We identify three phases having mostly distinct colors (φx), splitted by the two periods P3 mid 2001 and P8 mid 2003. At this periods large parts of file ownerships and commit colors change. In P3 we see the violet and navy blue files in S1 become red and yellow. The green author introduces a lot of new files, doubling the size of S1. A quick look at the commit messages tells us, that these new files were added to implement EJB 2.0, marking the beginning of the implementation of JBoss version 3. The other phase-split at P8 is branded by radical takeover of the EJB module S1. Nearly every file that was owned by the green author changes the ownership to cyan and magenta. Again looking at the commit messages, we discover that these takeovers are CVS merges to or from the branch of JBoss 4, predicting the new release version of the project.

Both periods are accompanied by Cleanings and Expansions. Also interesting is the fact, that most developers who lose ownership at one of these two periods, do not make any commits afterwards. Thus, without having spent much effort, we conclude that the composition of the JBoss team changed at major project releases, and the new teams restructured the system.

The project's hottest periods, the ones with the most commits and changes, are P3 and P4. During this time the system was wide Expanded and Cleaned, again pointing to large refactorings of the new team members.

In JBoss no evidence of a control about who may commit at which part of the system can be found. The developers make use of the freedom of committing where they need to change something. But, at the other hand we have many unicolored blocks, which in turn point to some sort of responsibility of the authors to a specific part of the project. This is also a common pattern found in modularized systems – *i.e.,* the system is split up into smaller components – where the developers are responsible for some modules but have the freedom to commit where they want.

There are no real Dialogs between the authors that last longer than just for two commits. This may point to weakness in team communication, perhaps because the core developers did not sit in the same room while working and communication over email (or chat) is not that efficient.

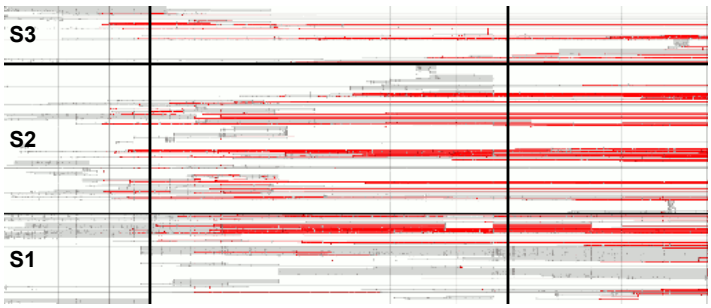
5.3.2 The Story of the Authors

In this section we look at some of the most important and interesting authors of JBoss. We ordered all authors according to their number of commits and picked 7 developers

from the top ten. For each of them we show a small version of the *Ownership Map* of JBoss and a table with measurements belonging to that developer. We give a description of each author's behavior and explain how to interpret these measurements in the *Ownership Map*.

The small *Ownership Map* displays only the files of the author who is in focus. All other colors are grayed out. The lines are rendered exaggerated to highlight the areas where the author owned the files. The file axis is divided into the three S1 to S3 main modules, like the *Ownership Map* in Figure 5.3 (p.37). But these three modules are again divided into several submodules forming a total number of 16. We took care that each of the submodules have about the same size and that the files in the same submodule belong to the same implementation topic. The time axis is divided into the same three phases $\varphi 1$ to $\varphi 3$ as for the story of the system of JBoss.

In the table beside the *Ownership Map* we list the proliferation and focus metrics. See Section 3.6 (p.20) for the definition of them. We calculate each measurement with the touched and owned files of the author. The touched files are all those that the author ever modified during the project. And the owned files are those that the author at least once got the ownership of. The proliferation and focus are both calculated over the 16 submodules. Thus, the proliferation can achieve a maximum value of 16 and the focus a maximum of 1. A summary of these measurements for all authors is also listed in Figure 5.11 (p.43).



Files	touched	owned
Count	1165	756
Proliferation	7.687	5.069
Focus	0.554	0.353

Figure 5.4: The work of the red author.

Red author. Red is working in all three modules, but mostly in S2, the core files. He started to work after one year and is still part of the JBoss team. His work is well distributed over the whole system. As he modified nearly half of all JBoss files, he also would reach a high focus if he had worked localized on the same part of the system. But a value of 0.5 is not that high and tells us that Red's work was distributed over the system. He took ownership of about a third of the system during his acting and a major part of these files did survive till today, making his

knowledge about these files even more valuable.

Characteristic for Red's behavior is that he always Familiarizes himself with the code he is working on. He never took over a part aggressively, but did mostly small commits. He is surely *the* allrounder in JBoss and the only core developer from the mid phase $\varphi 2$ (P3-P8) that is still part of the team.

Files	touched	owned
Count	423	317
Proliferation	2.747	2.004
Focus	0.503	0.513

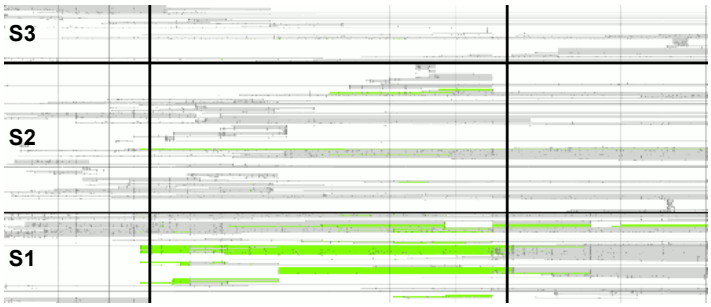


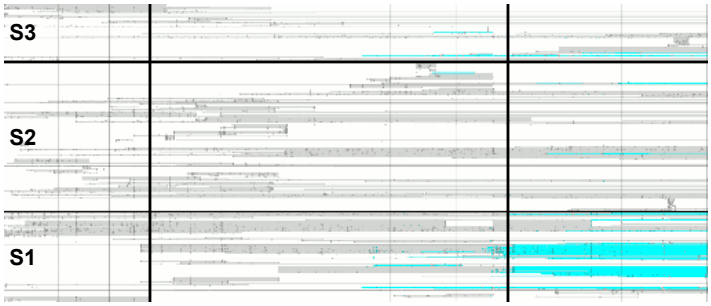
Figure 5.5: The work of the green author.

Green author. Green worked only during the mid phase $\varphi 2$ and mainly in S1. The large amount of the files he touched he also got to own. This means he either worked concentrated on the same files or he made large modifications. In Figure 5.5 (p.40) we see the *Ownership Map* and see that at the beginning of his work he radically Expanded the system introducing a new design and doubling the size of the module.

Looking at the metrics we see that his work is scarcely proliferated and his focus is high. This reflects in the *Ownership Map* view as large solid green blocks. We can also say that Green worked very concentrated on the same files during the whole phase $\varphi 2$.

As he hardly took over files from other authors he may be a developer who tends to rewrite code from scratch, rather than changing existing implementations. Most of the files he introduced are still part of the EJB implementation, and even though he does not own the module anymore, his knowledge about S1 is still valuable. He is probably the one to ask about the design of JBoss' EJB implementation, but he may not be comfortable with the rest of the system.

Cyan author. Cyan worked mainly in $\varphi 3$ on the EJB files S1. He can be seen as successor of Green. After intensive work in P7 he aggressively takes over the whole module, with a merge from the branch of JBoss 4 to the main branch. His focus is also same high as the one of the Green author. As he is the youngest owner of S1 he is the expert for this module.



Files	touched	owned
Count	400	286
Proliferation	2.523	1.866
Focus	0.492	0.488

Figure 5.6: The work of the cyan author.

Files	touched	owned
Count	371	105
Proliferation	2.508	0.729
Focus	0.236	0.078

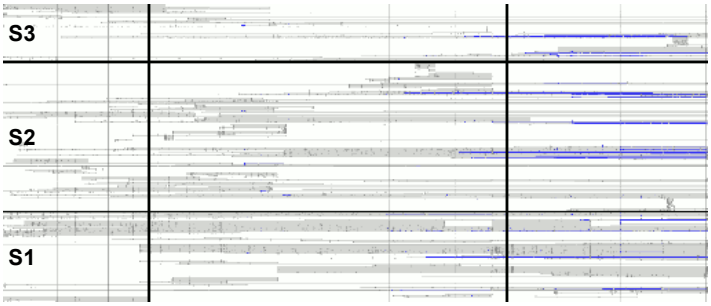
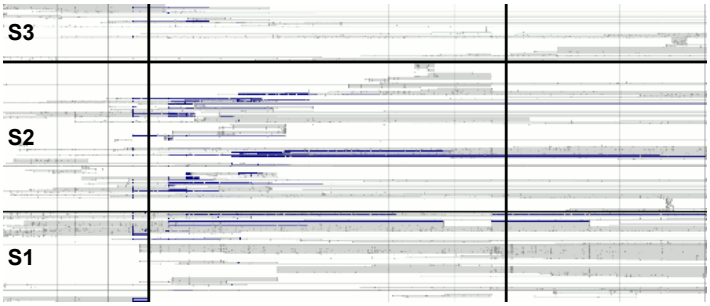


Figure 5.7: The work of the blue author.

Blue author. Blue worked from P4 to P10. Compared to the files he has touched, he owns only few files and those he owns show a low focus. His work is not localized, but distributed over the whole system. Looking at the *Ownership Map* in Figure 5.7 (p.41) we barely see any blue areas. Most of his commits are small, thus having no impact on the ownership.

In the commit messages the word "fixed" dominates the comments, classifying Blue as a pure Bugfix-author. Nevertheless, his knowledge about the system should not be underestimated. Unlike an Editor (see Navy author), fixing bugs requires deep understanding about the part to be fixed. The solution is usually small, explaining the small commits. Blue worked over 3 years fixing bugs in JBoss, thus he have accumulated broad knowledge of problematic parts in the system. Compared to Red, Blue may have deeper knowledge on specific parts, but Red will have the greater overview.

Navy author. Navy touched a lot of files in her short working period of nine months during P4 and P5. Despite her high value of proliferation, she has low focus. This



Files	touched	owned
Count	706	248
Proliferation	4.599	1.657
Focus	0.440	0.183

Figure 5.8: The work of the navy author.

means she really affected files across the whole system. In the visualization we see long vertical stripes of navy blue color, revealing Edits. To ensure the nature of Navy, we had a quick look at the commit message of the two largest commits, and see that they were non-functional changes to ensure coding guidelines. Navy is definitely an Editor.

Due to the short time she was acting in the project and the fact that reformatting code does not mean to understand it, Navy is not familiar with the system and her knowledge can be disregarded.

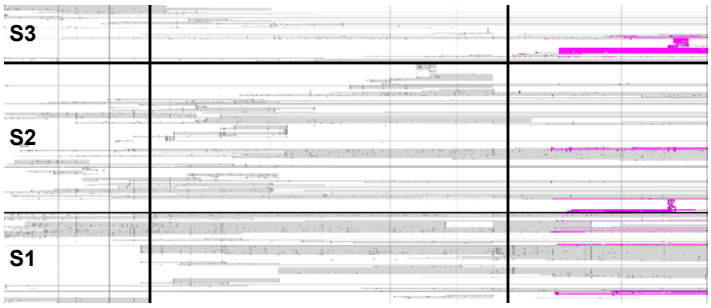
Files	touched	owned
Count	642	377
Proliferation	3.885	2.491
Focus	0.413	0.276

Figure 5.9: The work of the yellow author.

Yellow author. Yellow worked during P3 to P5, which was the most active period in JBoss. The focus of the owned files is low compared to the proliferation - her work is not localized, but spread over the system. And effectively in her *Ownership Map* in Figure 5.9 (p.42) we see wide distributed yellow lines.

She played a similar role as allrounder like Red and some of her files survived till today. But unlike Red, she left the project after this big refactoring phase. Her

knowledge is probably out of date.



Files	touched	owned
Count	389	288
Proliferation	2.591	1.988
Focus	0.585	0.574

Figure 5.10: The work of the magenta author.

Magenta author. Magenta started work after the big refactorings of S1 in $\varphi 3$. The focus of his work, both the modified and the owned files, is remarkably high. He worked very localized on the same files. In the visualization in Figure 5.10 (p.43) we see three magenta colored blocks, one in each module. In S1 he worked on ejbtimer, in S2 on the webservice folder and in S3 he created the jdk1.5 subdirectory. Everywhere he worked he expanded the system adding new files.

Concluding we can say that Magenta worked on new features for JBoss and he is the person to ask about these features as he worked alone.

Author	**Files Count**		**Proliferation**		**Focus**	
	touched	owned	touched	owned	touched	owned
Red	1165	756	7.687	5.069	0.554	0.353
Green	423	317	2.747	2.004	0.503	0.513
Cyan	400	286	2.523	1.866	0.492	0.488
Blue	371	105	2.508	0.729	0.236	0.078
Navy	706	248	4.599	1.657	0.440	0.183
Yellow	642	377	3.885	2.491	0.413	0.276
Magenta	389	288	2.591	1.988	0.585	0.574

Figure 5.11: Proliferation and focus measurements of the authors

5.3.3 Discussion

About Commit Regulations. In JBoss there is no control about who may commit at which part of the system. The developers make use of the freedom of committing where they need to change something. This lowers the focus of the files an author worked on.

About Focus. In JBoss, most of the core developers, even the most concentrated ones, hardly reach a focus of 0.5. In larger projects many files are created and deleted during the lifetime of the project. If a developer does not join the project from the beginning, he will not modify files that were already deleted nor will he modify files that are created after the developer left the project team. Hence, if we take all files of the system for the calculation into account the focus will take lower values. A solution would be to consider only those files that were part of the project's file base during the time the developer was acting.

About Change of Behavior. Instead of calculating the measurements for the whole project lifetime, the measurements can also be calculated for shorter time periods *e.g.,* 6 months. This would allow one to look for changing behavior of the authors. For example an increasing value for the focus would tell that the author concentrates his work more and more; perhaps because he took over the responsibility of a module.

5.4 Ant, Tomcat and JEdit

Figure 5.12 (p.45) shows the *Ownership Maps* of three open-source projects: Ant[2], Tomcat[3], and JEdit[4]. The views are plotted in the same way as the maps in the previous case studies. The only difference is that the vertical lines slice the time axis into periods of twelve months instead of three and respectively six months. Again, the files are ordered according to their commit signature.

Ant has about 4'500 files with 60'000 revisions, its repository spans six years. Tomcat has about 1'250 files and 13'000 revisions spanning over five years. And the CVS repository of JEdit contains about 500 files and 11'000 revisions and lasts a bit more than four years.

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

[2] http://ant.apache.org
[3] http://tomcat.apache.org
[4] http://www.jedit.org

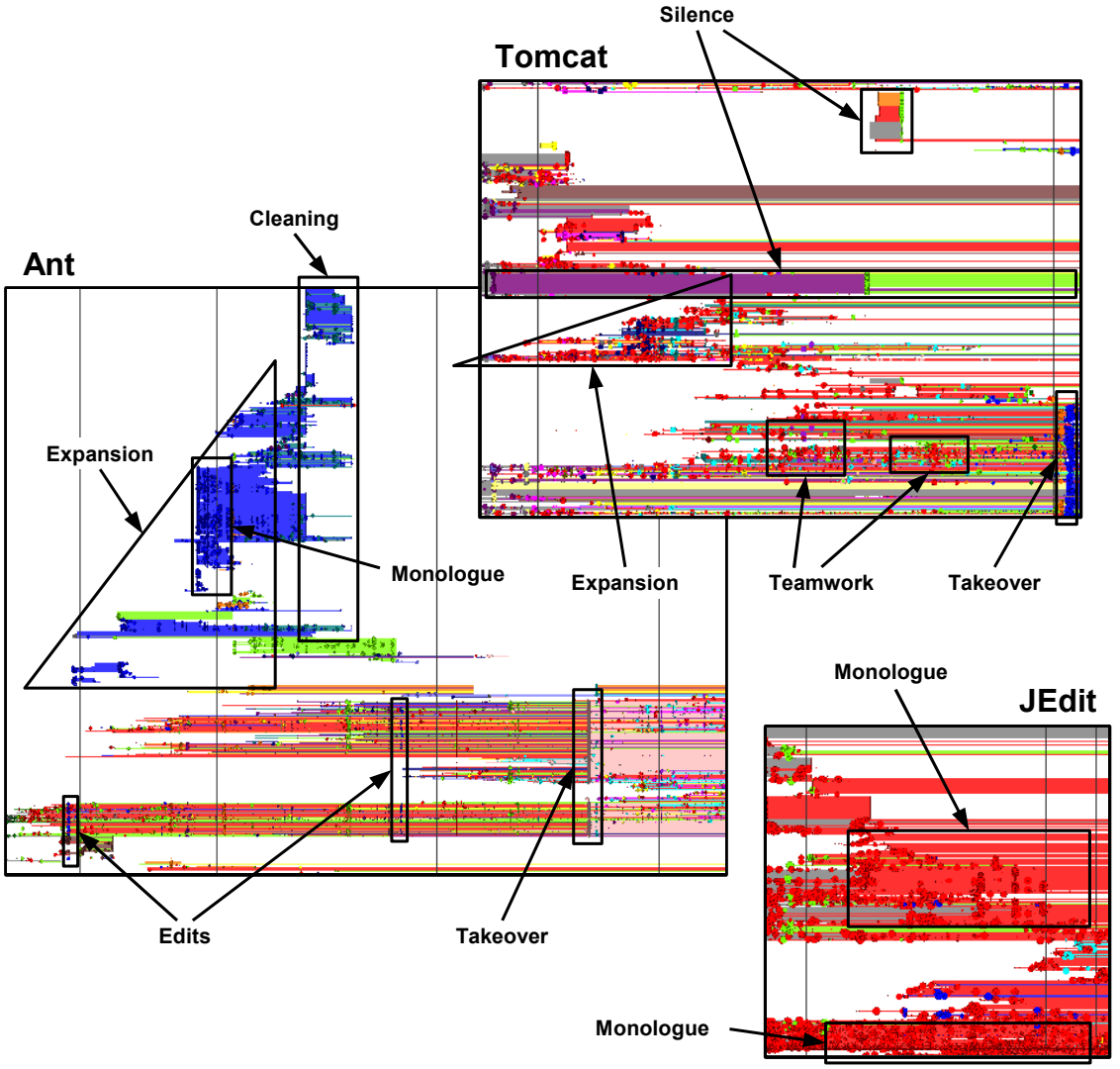


Figure 5.12: The *Ownership Maps* of Ant, Tomcat and JEdit.

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

We also see an aggressive Takeover by the pink author in the second last year of the project. The takeover may be the result of merges from another branch, marking a new Ant release. But as opposed to JBoss, see Section 5.3.1 (p.38), the team composition did not change that dramatically.

Tomcat. The colors in this view are, apart from some large blocks of Silence, well mixed. The *Ownership Map* shows much Dialogue and hotspots with Teamwork. Thus this project has developers that collaborate well. Here again, at the end of the bottom section blue starts a big takeover doing a lot of subsequent commits.

JEdit. This view is dominated by one sole developer, making him the driving force behind the project. This pattern is also often found in open-source projects: being the work of a single author who contributed about 80% of the code. But there is not any Dialogue either. The other authors work on their files and do not interact with each other.

Chapter 6

Chronia the Tool

6.1 Introduction

This chapter presents Chronia[1], the implementation of our approach in SmallTalk [Seeberger *et al.*, 2006]. It is built on top of the Moose[2] reengineering environment [Ducasse *et al.*, 2005].

Chronia uses a layered design and consists mainly of following three parts:

CVS client. A standalone CVS client implementation. It deals with most of the CVS pserver protocol and fetches versioning information directly out of a CVS repository.

Model, Moose layer. The implementation of the model, built on top of the Moose reengineering environment. This layer makes use of the own CVS client through a well defined interface that allows to implement access to other versioning systems.

Visualization. Interactive GUI part and visualization rendering layer based on the Model.

Structure of the Chapter

In Section 6.2 (p.48) we explain the core model of Chronia. Section 6.3 (p.49) shows some screenshots of Chronia in action presenting some of the features of our tool.

[1] http://www.coffeeoracle.ch/projects/chronia
[2] http://www.iam.unibe.ch/∼scg/Research/Moose

6.2 Model

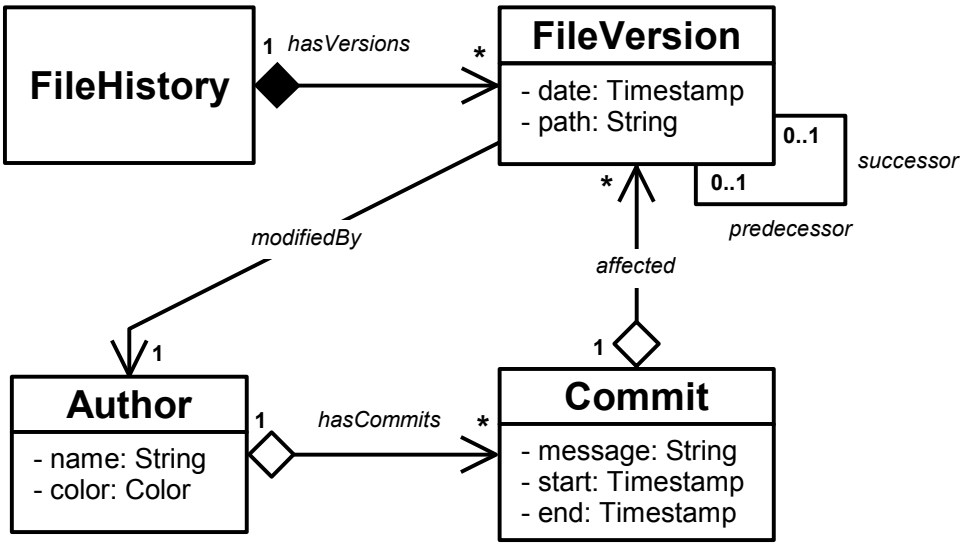


Figure 6.1: Concept of the core meta model of Chronia.

In Figure 6.1 (p.48) we see the core meta model of Chronia. This is only a concept of the core model, but it shows the essential information. It is designed following the concept of the Hismo meta-model [Gîrba, 2005]. The difference is that instead of having separate classes for version and snapshot (see Figure 2.1 (p.7)), we represent them as only one class named FileVersion.

The FileHistory is a first class entity representing the history of a file knowing all its file versions. The versions are ordered according their date and know their predecessor and successor and the author who modified the FileVersion. An Author has name and a color and knows its commits he has made to the system. A commit is a transaction that a user made to the versioning system. It knows its duration and which FileVersions were affected. Commits can have a message that a developer specified while committing the changes.

This model is built up from CVS, though a local file containing the CVS log is sufficient also. The entities follow the lazy initialization behavior, loading information from the CVS repository when needed.

Due to the integration of Chronia with the Moose reengineering environment, we profit of its meta model infrastructure. We can define relationships, action and properties for

this classes. This way we can interactively work with the model, *e.g.,* the history of a file.

6.3 View

Figure 6.2 (p.49) shows the main Moose interface containing the full history of the JBoss project. On the left we see a tree hierarchy view with the Moose entities grouped by type. We can manipulate them with mouse drag and drops and contextual menus. At the right side, Moose presents a list with the entities that are contained in the selected group. Moose also allows to select properties of the entities that should be displayed within this list. And on the bottom we have a tabbed pane providing several tools to filter and manipulate the entity groups.

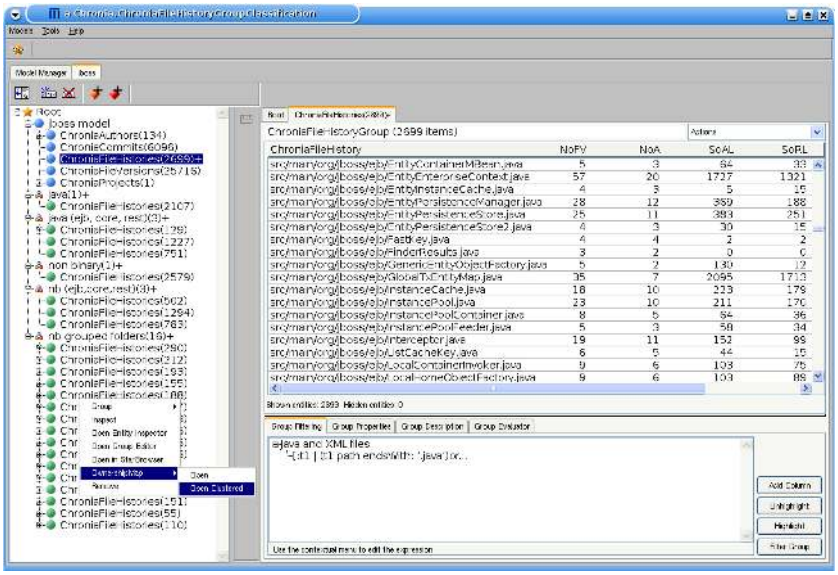


Figure 6.2: The history of JBoss loaded in Moose.

Using the contextual menu over a group, or grouped group, of file histories, commits or file versions, we can open the *Ownership Map* for this entities. Figure 6.3 (p.50) emphasizes the interactive nature of our tool. We see Chronia visualizing the overall history of the project, which provides a first overview. Since there is too much data we cannot give the reasoning only from this view, thus, Chronia allows for interactive zooming. For example, in the window on the lower right, we see Chronia zoomed into

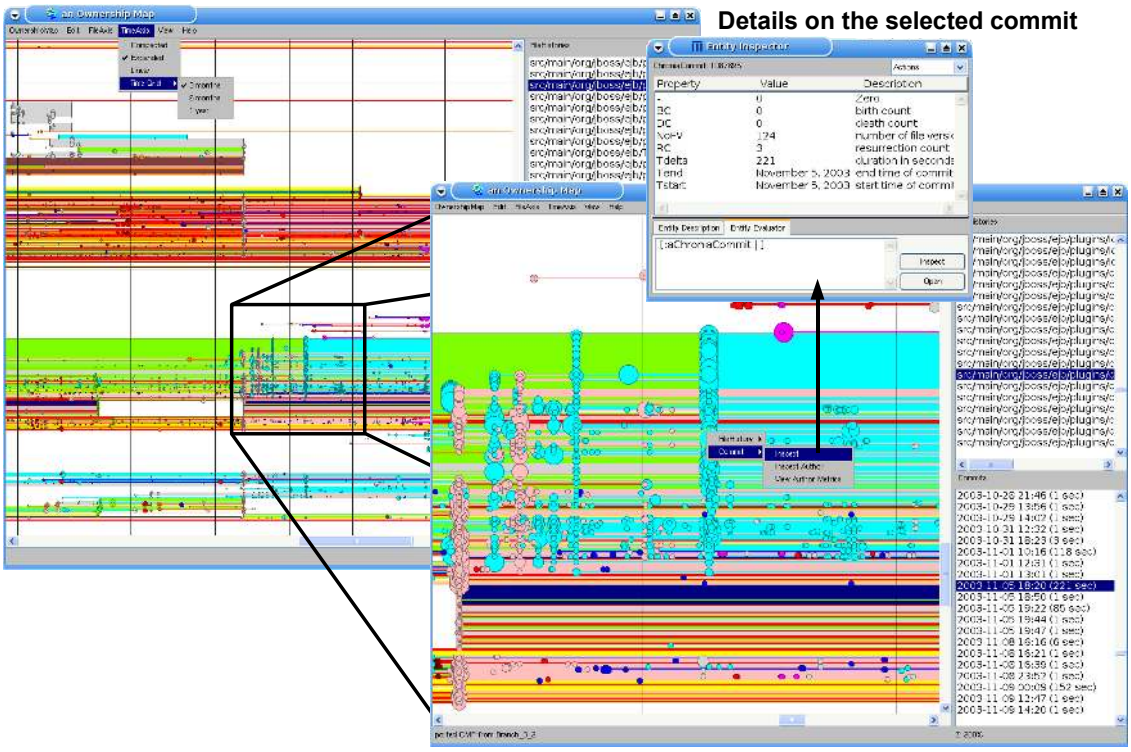


Figure 6.3: *Ownership Map*, zooming and accessing measurements.

the marked part of the original view. Furthermore, when moving the mouse over the *Ownership Map*, we complement the view by also showing the current position on both time and file axis highlighted in the lists on the right. These lists show all file names and the start timestamps and duration of all commits. Pointing with the mouse over a circle, the message of the commit – if any – is displayed at the bottom bar of the view.

As Chronia is built on top of Moose, it makes use of the Moose contextual menus to open detailed views on particular files, commits or authors. For example, in the top right window we see a view with metrics and measurements of a commit and an evaluation pane, that allows direct manipulation of this entity with SmallTalk code.

Chronia also allows the user to change rendering properties for both axes. On the top left window we see the opened menu for the time axis. There we can adjust the width of

the time grid to show a vertical line for example only every year. We can also tell Chronia to render the time in a linear way, making the distance between commits on the view to be proportional to their distance in time.

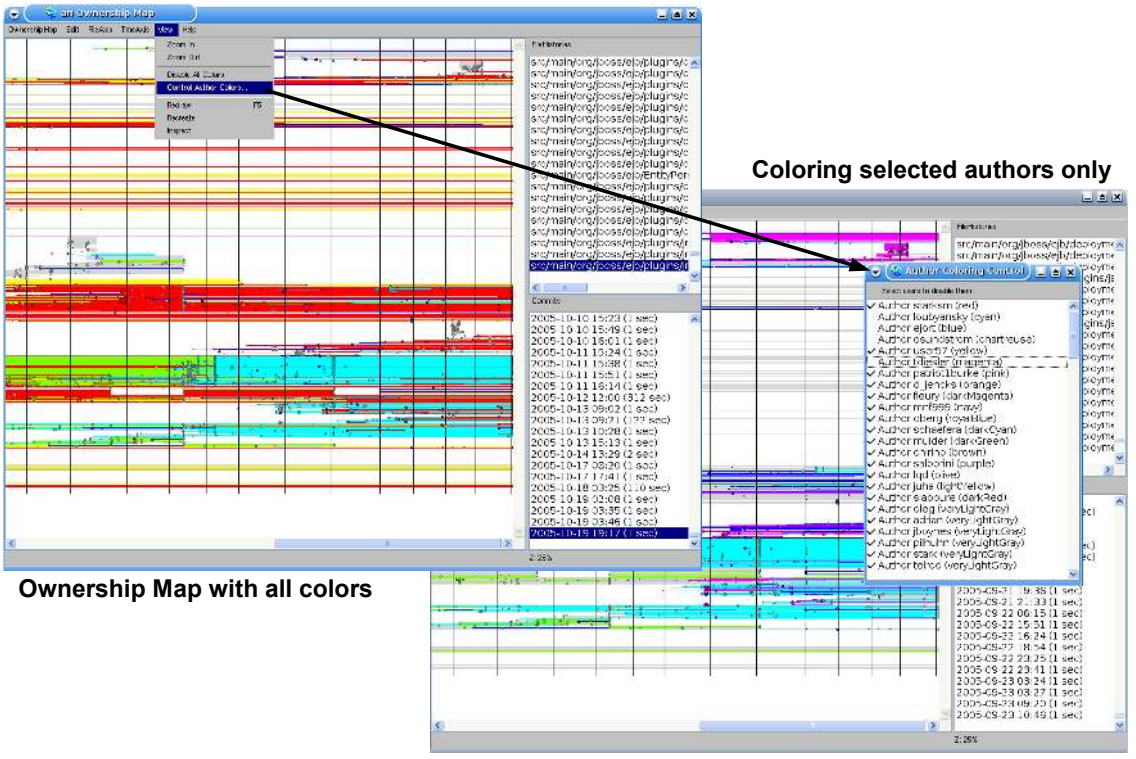


Figure 6.4: Enabling and disabling colors.

Colors are an important aspect in the *Ownership Map* visualization. Thus Chronia also allows us to enable and disable the colors of specific authors. This is particularly useful when the user is interested in the work of only one developer (see Section 5.3.2 (p.38)). In Figure 6.4 (p.51) we see on the left the visualization of JBoss, showing using all colors for all authors. Using the menu we can open a color control window where we can select the authors that should be rendered as gray. In the example we see at the right side the same *Ownership Map* with only four colors. As there is not anymore that much red, the view now better points out the work of the blue author.

Chapter 7

Conclusions

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

— How many authors developed the system?

— When and where worked the developers?

— What were the behaviors of the developers?

— Which author developed and owned which part of the system?

To answer them, we define the *Ownership Map* visualization based on the notion of code ownership and measurements. In addition we semantically group files that have a similar *commit signature* leading to a visualization that is not based on alphabetical ordering of the files but on co-change relationships between the file histories.

The *Ownership Map* and metrics that calculate the proliferation and focus of the author's work help in answering which authors are knowledgeable in which part of the system. Our approach of the visualization also reveals behavioral patterns that show how the authors developed the system. To show the usefulness we implemented the approach and applied it to several large case studies.

We did extensive experiments and reported the findings discussing different facets of the approach. We discussed the benefits and the limitations as we perceived them during the experiments.

7.1 Discussions

This section discusses open points and limitations of our approach and some difficulties we encountered while developing Chronia. First, we focus on topics about relying on versioning system information, particularly CVS. Second, we discuss issues related to the visualization.

7.1.1 About Versioning System Information

On the decision to rely on CVS log only

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

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

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

On the shortcomings of CVS as a versioning system

As CVS lacks support for true file renaming or moving, this information is not recoverable without time consuming calculations. To move a file, one must remove it and add it later under another name. Our approach identifies the author doing the renaming as the new owner of the file, where in truth she only did rename it. For that reason, renaming directories impacts the computation of code ownership in a way not desired.

CVS Branches

We do not treat CVS branches in a different way than the main branch. That is we just order the file versions according their modification date ignoring from which branch they are. This has an influence on the calculation of the ownership. For example, File X is owned by author A. Now, author B modifies the file to become File X' on a new branch and he also gets the ownership of this file. As author B continues to work on File X' it seems on the *Ownership Map* that author A "lost" the file ownership on File X and stopped to work on it, where in fact he would be the owner in the main branch.

This is only a problem for large branches that are not merged back into the main branch. After this merge author B becomes the owner of File X. A solution would be to identify large branches and to display them on their own on the *Ownership Map*. This way it would also be easier to identify release cycles as larger projects always use branches when the team starts to work on new releases.

Commit Culture

Not every developer has the same habit of committing his changes. Some authors commit every few changes, others develop the whole day and commit before finishing the work. Authors who commit more often make a higher noise. They commit code they change some hours later. This has an impact on the *Ownership Map* for examinations of short time periods, but will lose importance when analyzing the evolution over longer periods.

7.1.2 About the Visualization

On the scalability of the visualization

Although Chronia provides zooming interaction, one may lose the focus on the interesting project periods. A solution would be to further abstract the time and group commits to versions that cover longer time periods. The same applies to the file axis grouping related files into modules.

On the perspective of interpreting the *Ownership Map*

In our visualization we sought answers to questions regarding the developers and their behaviors. We analyzed the files from an author perspective point of view, and not from a file perspective of view. Thus the *Ownership Map* tells the story of the developers and not of the files *e.g.,* concerning small commits: subsequent commits by different authors to one file do not show up as a hotspot, while a commit by one author across multiple files does. The latter is the pattern we termed *Edit*.

Interpret with Care

From a project manager point of view the *Ownership Map* can give valuable hints. Knowing whether a developer tends more to Takeover or more to Familiarization is a good

indicator to whom the responsibility of the subsystem should be given. If a subsystem needs rewrite and restructuring the Takeover type is a good choice, otherwise if a subsystem is a good base to be built up on the Familiarization type is a good choice.

But classifications of the authors have to be interpreted in their context. If a developer aggressively takes over subsystems this does not mean that he has a bad character or that he will always tend to Takeovers. In our case study (Figure 5.2 (p.34)) Green's Takeover of S2 in P7 must be seen in the context of the system history: Blue left the team and Green was the original developer of S2. Maybe he would have acted differently if Blue were still in the team.

7.2 Future Work

The fields in which we could conduct further research in code change analysis focussing on author information is huge. Regarding just our approach, there are still open tasks we could explore.

Alternative Clustering

The order of the file axis in the *Ownership Map* is based on grouping files with similar commit signature, *i.e.,* files that changed together are positioned near each other. Suppose we focus on specific authors and want to ask: When did this author expand the system? At which period did he own the most files?

If we want to answer this sort of questions using the *Ownership Map* only, our clustering method is not optimal. For example, in the JBoss case study in Figure 5.4 (p.39) the Red author worked distributed over the system. It is hard to identify when he added files or owned the most files. Grouping the files according to their ownership signature, *i.e.,* files with same ownership history are near each other, we can better answer the above questions.

Other Abstraction Levels

In the future, we would like to investigate the application of the approach at other levels of abstractions besides files, *e.g.,* classes, methods, *etc.* This would require a more difficult preprocess *e.g.,* parsing the files.

Also interesting is to take into consideration types of changes beyond just the change of a line of code.

Accessing Other Versioning Systems

Our approach is tuned to work with CVS. But there are other versioning systems that are worth to be analyzed with our approach. Every versioning system has its own advantages and shortcomings and this would again allow respectively require some tuning of the approach. A common versioning system interface is needed, so that our approach can build upon it. Then we would only need to write a wrapper around the specific system providing the required interface. In fact our tool Chronia is implemented relying on such an interface. But this interface is tuned to support the file level abstraction only.

List of Figures

Bibliography

[Ball and Eick, 1996] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996. 10, 11

[Bertin, 1974] Jacques Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974. 24

[Capiluppi *et al.*, 2004] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. Evolution of understandability in OSS projects. In *Proceedings 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 58–66, Los Alamitos CA, 2004. IEEE Computer Society Press. 9

[Capiluppi, 2003] Andrea Capiluppi. Models for the evolution of OS projects. In *Proceedings International Conference on Software Maintenance (ICSM 2003)*, pages 65–74, Los Alamitos CA, 2003. IEEE Computer Society Press. 10

[Chuah and Eick, 1998] Mei C. Chuah and Stephen G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998. 10

[Conway, 1968] Melvin E. Conway. How do committees invent ? *Datamation*, 14(4):28–31, April 1968. 1

[Demeyer *et al.*, 2002] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. 1, 13

[Draheim and Pekacki, 2003] Dirk Draheim and Lukasz Pekacki. Process-centric analytical processing of version control data. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 131–136, Los Alamitos CA, 2003. IEEE Computer Society Press. 8

[Ducasse *et al.*, 2005] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005. 2, 4, 47

[Eick *et al.*, 2002] Stephen Eick, Todd Graves, Alan Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002. 9, 10

[Gall *et al.*, 1997] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM'97)*, pages 160–166, Los Alamitos CA, 1997. IEEE Computer Society Press. 9

[Gall *et al.*, 1998] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press. 12, 25

[Gîrba *et al.*, 2005a] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE)*, pages 113–122. IEEE Computer Society Press, 2005. 3

[Gîrba *et al.*, 2005b] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society. 9

[Gîrba, 2005] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005. 7, 48

[Godfrey and Tu, 2000] Michael Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings International Conference on Software Maintenance (ICSM 2000)*, pages 131–142, Los Alamitos CA, 2000. IEEE Computer Society Press. 9

[Hassan and Holt, 2004] Ahmed Hassan and Richard Holt. Predicting change propagation in software systems. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293, Los Alamitos CA, September 2004. IEEE Computer Society Press. 12

[Jain *et al.*, 1999] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999. 26

[Jazayeri, 2002] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technologies-Ada-Europe 2002*, pages 13–23, Berlin, 2002. Springer Verlag. 9

[Lanza and Ducasse, 2002] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In

Proceedings of Langages et Modèles à Objets (LMO 2002), pages 135–149, Paris, 2002. Lavoisier. 9, 24

[Lanza, 2003] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003. 9

[MacKenzie *et al.*, 2003] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd., 2003. 16

[Mockus and Votta, 2000] Audris Mockus and Lawrence Votta. Identifying reasons for software change using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 120–130. IEEE Computer Society Press, 2000. 8

[Mockus and Weiss, 2000] Audris Mockus and David Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2), April 2000. 8

[Mockus *et al.*, 1999] Audris Mockus, Stephen Eick, Todd Graves, and Alan Karr. On measurements and analysis of software changes. Technical report, National Institute of Statistical Sciences, 1999. 8

[Seeberger *et al.*, 2006] Mauricio Seeberger, Adrian Kuhn, Tudor Gîrba, and Stéphane Ducasse. Chronia: Visualizing how developers change software systems. In *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, 2006. 3, 47

[Tufte, 1990] Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990. 24

[Čubranić and Murphy, 2003] Davor Čubranić and Gail Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press. 12

[Čubranić, 2004] Davor Čubranić. *Project History as a Group Memory: Learning From the Past*. PhD thesis, University of British Columbia, Vancouver BC, December 2004. 12

[Van Rysselberghe and Demeyer, 2004] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, September 2004. IEEE Computer Society Press. 9, 25

[Viégas *et al.*, 2004] Fernanda Viégas, Martin Wattenberg, and Kushal Dave. Studying cooperation and conflict between authors with history flow visualizations. In *In Pro-*

ceedings of the Conference on Human Factors in Computing Systems (CHI 2004), pages 575–582, April 2004. 10, 13

[Voinea *et al.*, 2005] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 47–56, St. Louis, Missouri, USA, May 2005. 10

[Ware, 2000] Colin Ware. *Information Visualization.* Morgan Kaufmann, 2000. 24

[Wu *et al.*, 2004a] Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, November 2004. IEEE Computer Society Press. 10

[Wu *et al.*, 2004b] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99, Los Alamitos CA, November 2004. IEEE Computer Society Press. 10

[Zimmermann and Weißgerber, 2004] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press. 7

[Zimmermann *et al.*, 2003] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83, Los Alamitos CA, 2003. IEEE Computer Society Press. 12

[Zimmermann *et al.*, 2005] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *IEEE Transactions on Software Engineering*, June 2005. 12, 26