How Developers Drive Software Evolution

In proceedings of International Workshop on Principles of Software Engineering (IWPSE 2005)

Tudor Gîrba Adrian Kuhn Mauricio Seeberger Stéphane Ducasse

Software Composition Group
University of Berne, Switzerland
{girba, akuhn, mseeberg, ducasse}@iam.unibe.ch

Abstract

As systems evolve their structure change in ways not expected upfront. As time goes by, the knowledge of the developers becomes more and more critical for the process of understanding the system. That is, when we want to understand a certain issue of the system we ask the knowledgeable developers. Yet, in large systems, not every developer is knowledgeable in all the details of the system. Thus, we would want to know which developer is knowledgeable in the issue at hand. In this paper we make use of the mapping between the changes and the author identifiers (e.g., user names) provided by versioning repositories. We first define a measurement for the notion of code ownership. We use this measurement to define the Ownership Map *visualization to understand when and how different developers interacted in which way and in which part of the system*[1]*. We report the results we obtained on several large systems.*

Keywords: software evolution, software visualization, reverse engineering, development process

1 Introduction

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

From another perspective, Conway's law [4] 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 [5].

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

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

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

Contrary to similar approaches [14], 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 [8].

We implemented our approach in Chronia, a tool built on top of the Moose reengineering environment [6]. As CVS is a de facto versioning system, our implementation relies on the CVS model. Our aim was to provide a solution that gives fast results, therefore, our approach relies only on information from the CVS log without checking out the whole repository. As a consequence, we can analyze large systems in a very short period of time, making the approach usable in the early stages of reverse engineering.

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

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

The contributions of the paper are:

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

The paper develops as follows. In Section 2 we define how we measure the code ownership. In Section 3, we use this measurement to introduce our *Ownership Map* visualization of how developers changed the system. Section 4 shows the results we obtained on several large case studies, and Section 5 discusses the approach including details of the implementation. Section 6 presents the related work. We conclude and present the future work in Section 7.

2 Data Extraction from CVS log

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

The straightforward approach is to checkout all file versions ever committed to the versioning repository and to compute the code ownership from diff information between each 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.

In this paper, we aim to provide for an approach that can deal with large projects with long history, and that can provide the results fast. As CVS is the very common versioning system, we make tuned our approach to work with the information CVS can provide. In particular we compute the ownership of the code based only on the CVS log information.

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

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

2.1 Measuring File Size

Let s_{f_n} be the size of revision f_n, measured in number of lines. The number of lines is not given in the CVS log, but can be computed from the deltas a_{f_n} and r_{f_n} of added and removed lines. Even though the CVS log does not give the initial size s_{f_0}, we can give an estimate based on the fact that one cannot remove more lines from a file than were ever contained. We define s_{f_n} as in Figure 1: we first calculate the sizes starting with an initial size of 0, and then in a second pass adjust the values with the lowest value encountered in the first pass.

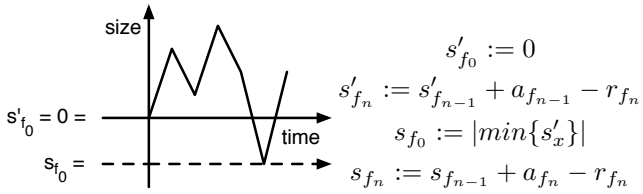


Figure 1. The computation of the initial size.

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

2.2 Measuring Code Ownership

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

Let $own^{\alpha}_{f_n}$ be the percentage of lines in revision f_n owned by author α. Given the file size s_{f_n}, and both the author α_{f_n} that committed the change and a_{f_n} the number of lines he added, we defined 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 the definition we assume that the removed lines r_{f_n} are evenly distributed over the ownership of the predecessor developers f_{n-1}.

3 The Ownership Map View

We introduce a the *Ownership Map* visualization as in Figure 2. The visualization is similar to the Evolution Matrix [12]: 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 that 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. And the color of the line denotes the author who owns most of the file.

Bertin [2] 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 [13], and Ware [16] - *e.g.,* we take into account that the human brain is capable of processing fewer than a dozen distinct colors.

In a large system, we can have hundreds of developers. Because the human eye is not capable of distinguishing that many colors, we only display the authors who committed most of all changes using distinct colors; the remaining authors are represented in gray. Furthermore, we also represent with gray 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 that performed the import. In short, a gray line represents either an unknown owner, or an unimportant one.

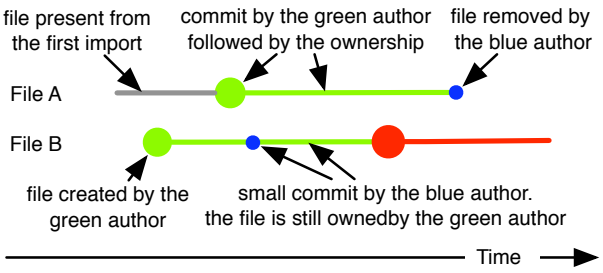


Figure 2. Example of ownership visualization of two files.

In the example from Figure 2, each line represents the lifetime of a file; each circle represents a change. File A appears gray in the first part as it originates from the initial import. Later the green author significantly changed the file, and he became the owner of the file. In the end, the blue author deleted the file. File B was created by the green author. Afterwards, the blue author changed the file, but still the green author owned the larged 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.

3.1 Ordering the Axes

Ordering the Time Axis. Subsequent file revisions committed by the same author are grouped together to form a transaction of changes *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 [20]. The difference is that in our approach the revisions in a commit do not have to have the same log comment, thus any quick subsequent revisions by the same author are grouped into one commit.

Ordering the Files Axis. A system may contain thousands of files; furthermore, an author might change multiple files that are not near each other if we would represent the files in an alphabetical order. Likewise, it is important to keep an overview of the big parts of the system. Thus, we need an order that groups files with co-occurring changes near each other, while still preserving the overall structure of the system. To meet this requirement we split the system into high-level modules (*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 [2] $\delta(H_a, H_b)$:

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

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

With this metric we can order the files according to the result of a hierarchical clustering algorithm [10]. 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 paper are ordered with *average linkage* clustering and *larger-clusters-first* tree traversal. Nevertheless, our tool Chronia allows customization of the ordering.

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

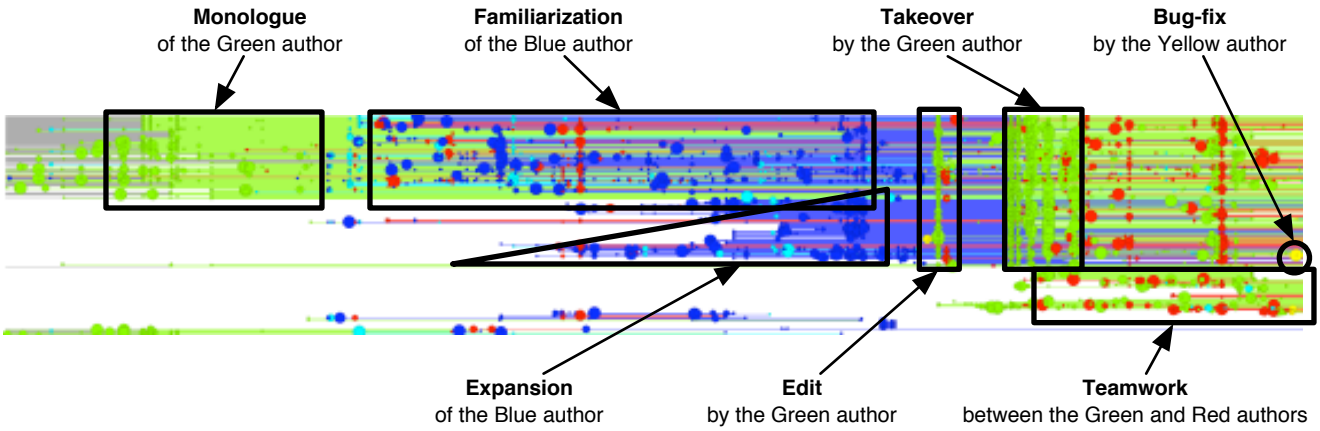


Figure 3. Example of the Ownership Map view. The view reveals different patterns: Monologue, Familiarization, Edit, Takeover, Teamwork, Bug-fix.

3.2 Behavioral Patterns

The Overview Map reveals semantical information about the work of the developer. Figure 3 shows a part of the *Ownership Map* of the Outsight case study (for more details see Section 4.1). In this view we can identify several different behavioral patterns of the developers:

- *Monologue.* Monologue denotes a period where all changes and most files belong to the same author. It shows on a *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 a *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 a *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 a *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 a *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 that removes a part of the system. We do not see this behavior in the example.
- *Bugfix.* By bug fix we denote a small, localized change that does not affect the ownership of the file. On a *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 a *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*.

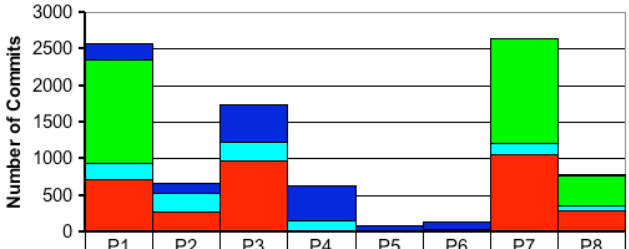
4 Validation

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

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

Open-source Case Studies. We choose Ant, Tomcat, JEdit, and JBoss to illustrate different fingerprints systems can have on an *Ownership Map*. Ant has about 4500 files, Tomcat about 1250 files, JEdit about 500 files, and JBoss about 2000 files. The CVS repository of each project goes back several years.

4.1 Outsight



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

Figure 4. Number of commits per team member in periods of three months.

The first step to acquire an overview of a system is to build a histogram of the team to get an impression about the fluctuations of the team members over time. Figure 4 shows that a team of four developers is working on the system. There is also a fifth author contributing changes in the last two periods only.

Figure 5 shows the *Ownership Map* of our case study. The upper half are Java files, the bottom half are JSP pages. The files of both modules are ordered according to the similarity of their commit signature. For the sake of readability we use S1 as a shorthand for the Java files part of the system, and S2 as a shorthand for the JSP files part. Time is cut into eight periods P1 to P8, each covering three months. The paragraphs below discuss each period in detail, and show how to read the *Ownership Map* in order to answer our initial questions.

The shorthands in paranthesis denote the labels R1 to R15 as given on Figure 5.

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

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 untill 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 4 identifies Red as the main contributor. The Expansion of S1 evolves in sudden steps (R9), and as their file base grows the Teamwork between Red and Cyan becomes less tight. In 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 4 shows that Red left the team, which consists now of Cyan and Green only. Cyan acts as an allrounder providing changes to both S1 and S2, and Blue is further working on S2. The work of Blue culminates in an Epilogue marking the end of this period (R8). He has now completely taken over ownership of S2, while the ownership of subsystem S1 is shared between Red and Cyan.

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

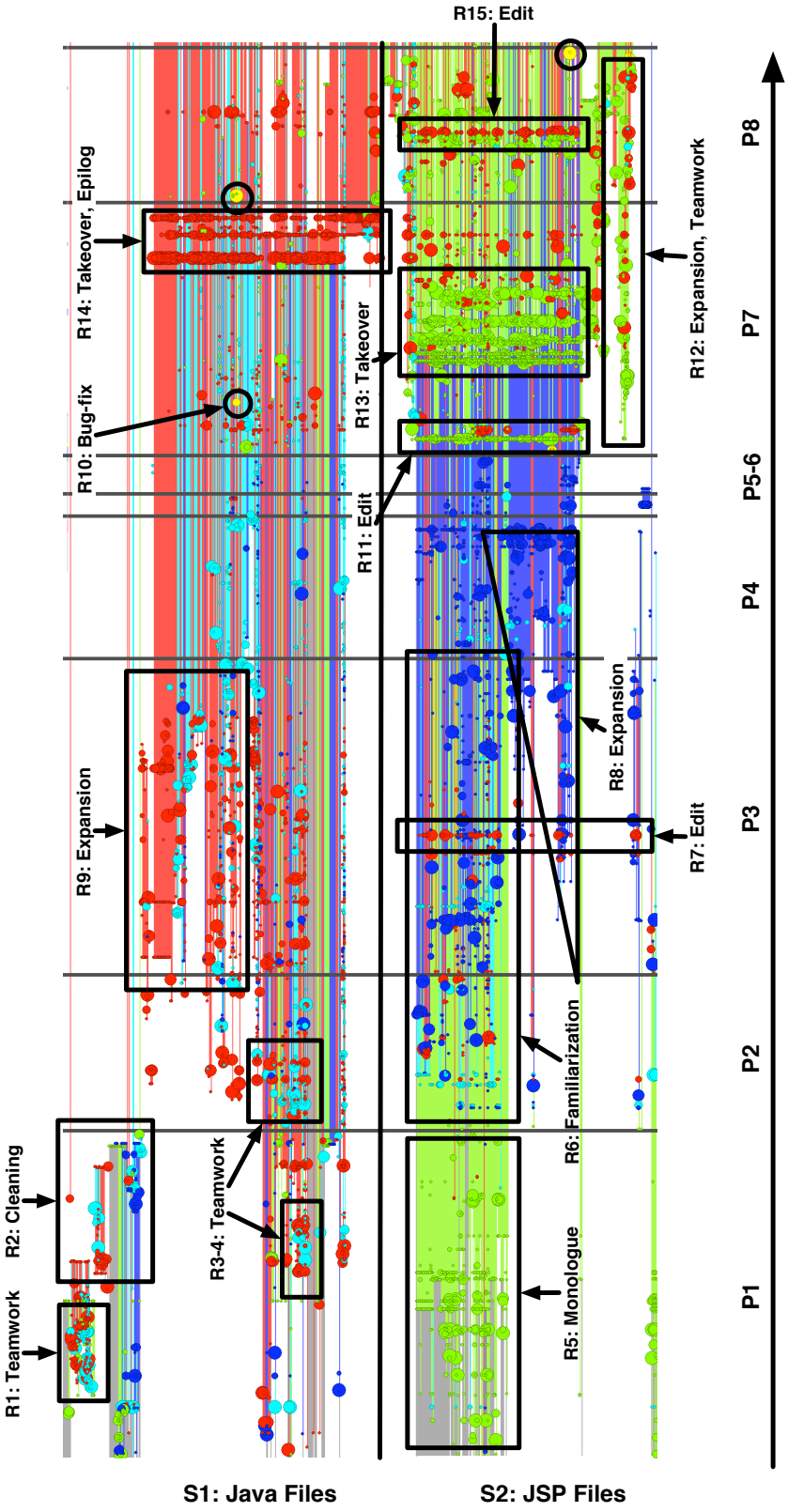


Figure 5. The Ownership Map of the Outsight case study.

by author Blue.

Period 7. After two periods of maintenance the team resumes work on the system. In Table 4 we see how the composition of the team changed: Blue leaves and Green comes back. Green restarts with an Edit in S2 (R11), later followed by a quick sequence of Takeovers (R13) and thus claiming back the ownership over his former code. Simultaneous he starts expanding S2 in Teamwork with Red (R12).

First we find in S1 selective changes by Red and Cyan scattered over the subsystem, followed by a period of Silence, and culminating in a Takeover by Red in the end *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.

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

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

Cyan author. Cyan is the only developer that 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.

4.2 Ant, Tomcat, JEdit and JBoss

Figure 5 shows the *Ownership Map* of four open-source projects: Ant, Tomcat, JEdit, and JBoss. The views are plotted with the same parameters as the map in the previous case study, the only difference is that vertical lines slice the time axis into periods of twelve instead of three months. Ant has about 4'500 files with 60'000 revisions, Tomcat about 1'250 files and 13'000 revisions, JEdit about 500 files and 11'000 revisions, and JBoss about 2'000 files with 23'000 revisions.

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

Ant. The view is dominated by a huge Expansion. After some time of development, the very same files fall victim to a huge Cleaning. This pattern is found in many open-source projects: Developers start a new side-project and when grown up it moves to an own repository, or the side-project ceases and is removed from the repository. In this case, the spin-off is the ceased Myrmidon project, a development effort as potential implementation of Ant2, a successor to Ant.

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

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

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

5 Discussion

On the exploratory nature of the implementation. We implemented our approach in Chronia, a tool built on top of the Moose reengineering environment [6]. Figure 7 emphasizes the interactive nature of our tool.

On the left of Figure 7 we see Chronia visualizing the overall history of the project, which provides a first overview. Since there is too much data we cannot give the reasoning only from this view, thus, Chronia allows for interactive zooming. For example, in the window on the lower right, we see Chronia zoomed into the bottom right part of

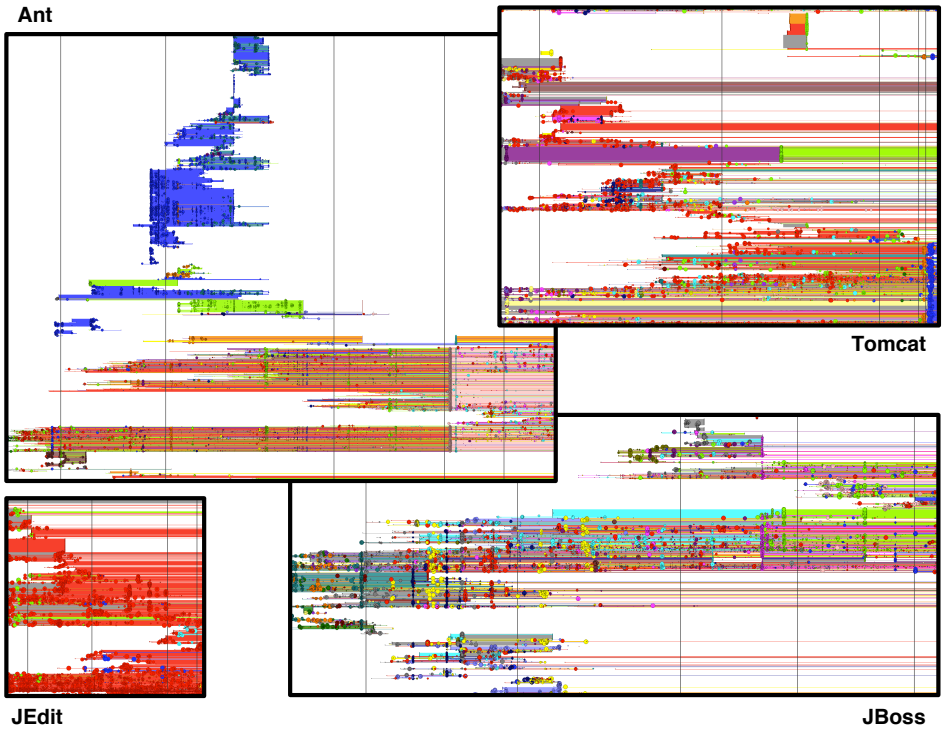


Figure 6. The Ownership Map of Ant, Tomcat, JEdit, and JBoss.

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 are highlighted in the lists on the right. These lists show all file names and the timestamps of all commits. As Chronia is build on top of Moose, it makes use of the Moose contextual menus to open detailed views on particular files, modules or authors. For example, in the top right window we see a view with metrics and measurements of a file revision.

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

On the decision to rely on 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 metainformation is required and not the source code itself. We consider this an advantage of our approach.

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

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 author to one file do not show up as a hotspot, while a commit by one author across multiple files does. The later being the pattern we termed *Edit*.

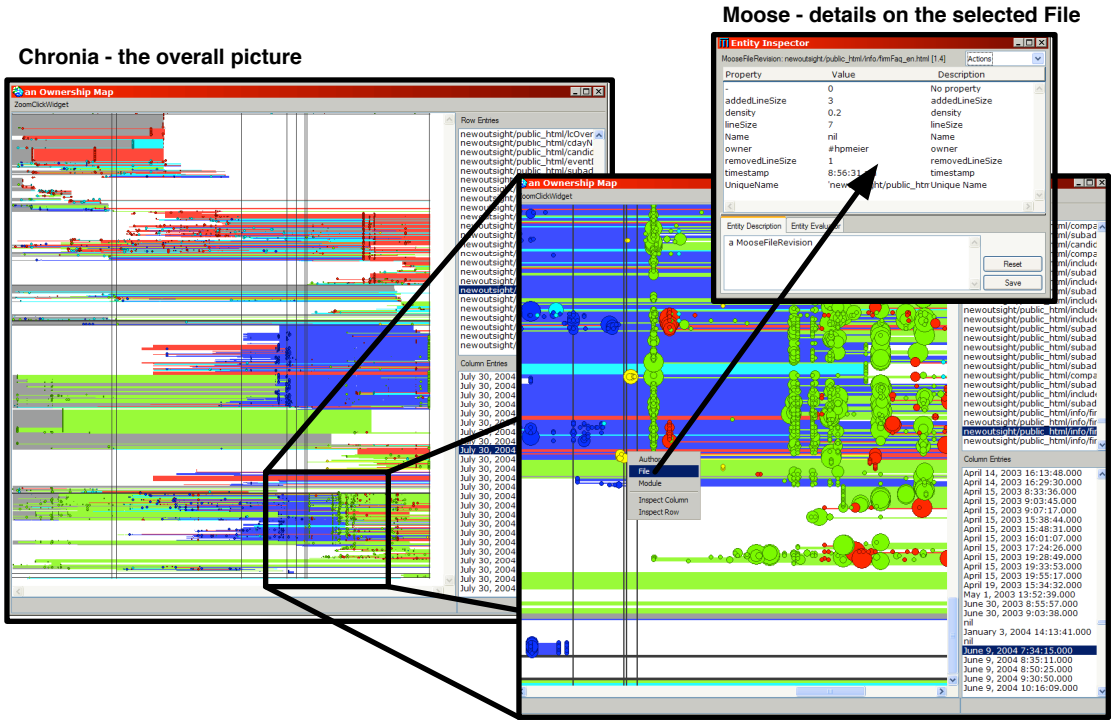


Figure 7. Chronia is an interactive tool.

6 Related Work

Analyzing the way developers interact with the system has only attracted few research. A visualization similar to *Ownership Map* is used to visualize how authors change a wiki page [15].

Xiaomin Wu *et al.* visualize [18] the change log information to provide an overview of the active places in the system as well as of the authors activity. They display measurements like the number of times an author changed a file, or the date of the last commitment.

Measurements and visualization have long been used to analyze how software systems evolve.

Ball and Eick [1] developed multiple visualizations for showing changes that appear in the source code. For example, they show what is the percentage of bug fixes and feature addition in files, or which lines were changed recently.

Eick *et al.* proposed multiple visualizations to show changes using colors and third dimension [7].

Chuah and Eick proposed a three visualizations for comparing and correlating different evolution information like the number of lines added, the errors recorded between versions, number of people working etc. [3].

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[14].

Jingwei Wu *et al.* use the spectrograph metaphor to visualize how changes occur in software systems [17]. They used colors to denote the age of changes on different parts of the systems.

Jazayeri analyzes the stability of the architecture [11] by using colors to depict the changes. From the visualization he concluded that old parts tend to stabilize over time.

Lanza and Ducasse visualize the evolution of classes in the Evolution Matrix [12]. Each class version is represented using a rectangle. The size of the rectangle is given by different measurements applied on the class version. From the visualization different evolution patterns can be detected such as continuous growth, growing and shrinking phases etc.

Another relevant reverse engineering domain is the analysis of the co-change history.

Gall *et al.* aimed to detect logical coupling between parts of the system [8] by identifying the parts of the system which change together. They used this information to define a coupling measurement based on the fact that the more times two modules were changed at the same time, the more they were coupled.

Zimmerman *et al.* aimed to provide mechanism to warn

developers about the correlation of changes between functions. The authors placed their analysis at the level of entities in the meta-model (*e.g.,* methods) [20]. The same authors defined a measurement of coupling based on co-changes [19].

Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graph [9].

7 Conclusions

In this paper we aim to understand how the developers drove the evolution of the system. In particular we ask the following questions:

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

To answer them, we define the *Ownership Map* visualization based on the notion of code ownership. 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* helps in answering which authors is knowledgeable in which part of the system and also reveal behavioral patterns. To show the usefulness we implemented the approach and applied it on several case studies. We reported some of the findings and we discussed the benefits and the limitations as we perceived them during the experiments.

In the future, we would like to investigate the application of the approach at other levels of abstractions besides files, and to take into consideration types of changes beyond just the change of a line of code.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006). We also thank AIESEC in Switzerland for allowing us to analyze the Outsight case study.

References

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[2] J. Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

[3] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, pages 24–29, July 1998.

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

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[6] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005.

[7] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *Software Engineering*, 28(4):396–412, 2002.

[8] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[9] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293. IEEE Computer Society Press, Sept. 2004.

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

[11] M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[12] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[13] E. R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[14] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004. to appear.

[15] F. Viégas, M. Wattenberg, and K. Dave. Studying cooperation and conflict between authors with history flow visualizations. In *In Proceedings of the Conference on Human Factors in Computing Systems (CHI 2004)*, pages 575–582, Apr. 2004.

[16] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[17] J. Wu, R. Holt, and A. Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89. IEEE Computer Society Press, Nov. 2004.

[18] X. Wu, A. Murray, M.-A. Storey, and R. 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. IEEE Computer Society Press, Nov. 2004.

[19] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83, 2003.

[20] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.