Visual Data Mining in Software Archives To Detect How Developers Work Together

Peter Weißgerber Mathias Pohl Michael Burch
Computer Science Department
University of Trier
54286 Trier, Germany
{weissger, pohlm, burchm}@uni-trier.de

Abstract

Analyzing the check-in information of open source software projects which use a version control system such as CVS or SUBVERSION can yield interesting and important insights into the programming behavior of developers. As in every major project tasks are assigned to many developers, the development must be coordinated between these programmers.

This paper describes three visualization techniques that help to examine how programmers work together, e.g. if they work as a team or if they develop their part of the software separate from each other. Furthermore, phases of stagnation in the lifetime of a project can be uncovered and thus, possible problems are revealed.

To demonstrate the usefulness of these visualization techniques we performed case studies on two open source projects. In these studies interesting patterns of developers' behavior, e.g. the specialization on a certain module can be observed. Moreover, modules that have been changed by many developers can be identified as well as such ones that have been altered by only one programmer.

1. Introduction

With the appearance of open source projects the complete evolution history of large software projects have become publicly available. In software archives, such as CVS and SUBVERSION, all revisions of all files that have ever existed during the evolution of a software are stored. Particularly, it is documented for each revision of a file when it has been *checked-in* and by which developer. The explanatory power of CVS data is also improved by the fact that mostly the developers of an open source project are locally separated. Thus, there often is no direct communication between the developers.

During the last years, information retrieved from software archives have been used by various researchers for many different interesting analyses (see Section 7). In our earlier work, we analyzed which software artifacts have been changed together (e.g. to recommend further changes [18]). In this work, we additionally examine *which* developers change the artifacts, and *when*. This implies the following questions:

- Is there one or more main developer(s), or rather is the work divided equally to multiple programmers? In the case that there are more main developers: Is the role of the main developer occupied by various developers during the lifetime of the project?
- Does each developer work on her own files and modules, or are there files and modules that are being worked on by multiple developers?
- Are there phases during the evolution, when there is a very active development, and such ones when there is hardly any development.

A common problem for (nearly) all kinds of analysis of software archives is to cope with the large amount of data in these archives. One approach is to use appropriate visualizations (which are often developed exactly for one particular analysis) that allow to navigate through the data in order to find interesting patterns.

We use three different visualization techniques to find answers to the questions asked above. In Sections 2 until 4 we present these techniques in detail. Next, in Section 5 we show by the means of case studies on JUNIT and TOMCAT3, how these techniques supplement each other in finding answers to these questions. Additionally, we present our results in that section.

As our analysis works on the data retrieved from version control systems, Section 6 discusses additional data sources.

Section 7 gives an overview of the related work and possible future work, while Section 8 concludes and summarizes this paper.

2. Transaction Overview

The whole set of transaction data is too large to analyze it directly. Thus, at first we need an overview of the information. The graphical representation of the overview should show a maximum of the data without hiding too many information (which is often a problem with visualization techniques).

The transaction overview is such a kind of visualization technique (Figure 1). It represents a coordinate system, in which the time axis is shown at the x-axis. The point at the leftmost position stands for the starting point of the software project, whereas the rightmost point shows the current or the last committed transaction in the evolution process of the system. The y-axis gives information about how many files have been changed in the corresponding transaction. A vertical line represents such a transaction at this special point in time. The value on the y-axis of the corresponding colored point signalizes the number of changes of the developer who did this check-in. Each developer is mapped to a unique color. Moreover, each vertical line is marked at the position of the y-axis where the corresponding files are located.

Generally there are five different aspects that can be detected using the transaction overview:

- **Number and frequency of the transactions:** A software system normally does not evolve linearly. That means, that there are phases, in which many check-ins are performed, whereas the project seems to be stall in other phases. Die density of the transactions in one special time interval is indicated by the frequency of the vertical lines. The bigger the distance between the lines the more time has been passed between these consecutive transactions. The existence of many vertical lines indicates that there have been many transactions within a short time interval.

- **Number of developers:** The number of developers is visualized in the number of different point colors.

- **Number of changed files in one single transaction:** The higher the point is located on the y-axis, the more files this developer changed in this transaction. This height is always related to the number of changed files in the biggest transaction.

- **Hierarchy-level of the changed file:** In earlier work [4] we analyzed, if there have been common changes between several hierarchy levels and we called these phenomena outliers or anomalies. To detect these outliers we sort the files hierarchically at the y-axis. [1]

- **Sequence of developers that are responsible for the changes:** Patterns in the change sequence of the developers can be detected with different color sequences at the x-axis.

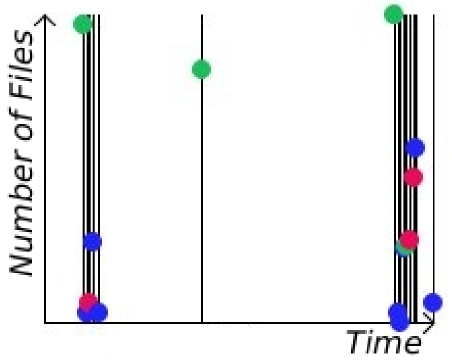


Figure 1. Small example of the overview of transactions visualization technique

3. File Author Matrix

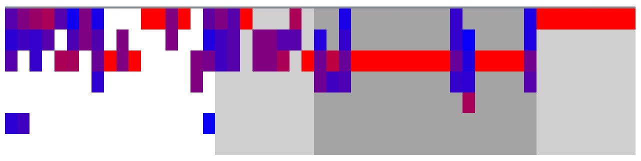


Figure 2. Example of a file author matrix

Figure 2 shows an example of the file author matrix, which is a space filling two-dimensional visualization. The x-axis contains the files, and the y-axis contains the developers of the project under examination. The color of each pixel of the matrix indicates how often the corresponding file has been changed by the respective developer, relative to the total number of changes to that file. In the used color scale a blue pixel means that there are relatively few changes while a red pixel indicates many changes. If a file has not been changed at all by a developer, the corresponding pixel is drawn in the background color.

[1]This means, that files which are located in the same directory or subdirectory are located very close to each other at the axis. In the same subdirectory they are sorted alphabetically.

Obviously, the order of the files and of the developers on the axes is very important in this visualization. Thus, we sort the developers descending by the number of their check-ins. This, for example, helps to recognize, whether a particular file has been changed primarily by the main developer or rather by such a developer who only changes few files (e.g., because he works only on a particular module).

In addition to this, the files on the x-axis are sorted hierarchically. This is meaningful because we assume that files in the same directory are related to each other in some way. For example in JAVA files in the same directory usually belong to the same JAVA package, and the files in subdirectories belong to sub-packages. But also non-source-code files are often arranged as modules and each module is represented by a separate directory.

To be able to distinguish the hierarchy (directory) levels from each other, we set the background color for each file (e.g. the pixels corresponding to that file and developers that have not changed the file) according to a linear color scale from black to white: The deeper the file is located in the directory tree, the darker is the selected background color[2]. The main intention is to makes it easier to find out whether a developer only changes files in the same directory (including sub-directories) or if he rather changes files from many directories, it has one major drawback: for one directory all files in (maybe multiple) sub-directories are colored the same, because they have the same depth in the tree. Another problem occurs when a file has been changed by all developers: Then all pixels for that file are colored and the background color is not observable any more. To solve these issues, we use tool tips that dynamically show which file (including the directory) is currently selected with the mouse.

In most cases, a software project contains by far more files than developers. As a consequence, we get very wide matrixes. Thus, we continue drawing of the matrix in the next screen line when there is not enough horizontal space on the screen. By this, we get a space filling visualization which enables to visualize even large projects such as TOMCAT3 (2297 files and 40 developers) on one screen page.

4. Dynamic Author-File Graph

The structure among the developers can be analyzed by the *Dynamic Author File Graph* (AFG). An AFG consists of an ordered sequence of bipartite graphs containing information about which file has been changed by which developer during a certain period of time. Formally an AFG can be described as a n-tuple:

$$\begin{aligned} \text{AFG} &:= (g_0, \ldots, g_{n-1}) \text{ where} \\ \forall i \in \{0, \ldots, n-1\}: \quad g_i &:= (D_i \cup F_i, E_i) \text{ with} \\ & \quad\; E_i \subseteq D_i \times F_i \end{aligned}$$

D_i denotes the set of all developers that commit changes during period i and F_i denotes the set of files that have been changed during that period. The edge set of a graph g_i is defined as follows:

$$\begin{aligned} E_i &:= \{(d,f) \in D_i \times F_i \mid \text{Developer } d \\ & \qquad \text{changed the file } f \text{ during period } i\} \end{aligned}$$

There is no recommendation for the time ranges of each graph. Depending on the agility of the development process of a software either shorter periods (e.g. one week) as well as longer terms (e.g. one month) are valid. In our examples we studied the behavior of the development process using AFGs with one month per graph. Developers are depicted by circles with a large diameter whereas files are represented by nodes with a small diameter.

An AFG can be drawn for a project using a sequence of node-link diagrams (as shown in Figure 3). The sequence is interactively displayed step-by-step with animated changes of the graphs. The layout of the graphs is computed using the extended Foresighted Layout by Diehl et al. [7, 11]. This allows for the preservation of the user's mental map when viewing the sequence in an animation.

As the AFGs in the examples presented here usually contain many graphs, only short excerpts of them are presented and discussed. Besides, only the most important nodes are labelled in this work in order to keep the static pictures readable.

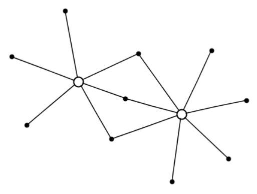


Figure 3. Example of a graph in an AFG sequence

5. Case studies

In the next subsections we apply the presented visualization techniques to CVS data obtained from two projects: a) the testing framework JUNIT, and b) the application server TOMCAT3. JUNIT has been created by 7 developers and has a size of 629 files and 522 transactions. TOMCAT3 is a

[2]Note, that we only used the first half of the color scale in our examples to prevent the pictures from getting too dark.

considerably larger project, containing 2297 files and 3917 transactions. 40 developers changed the CVS data of TOMCAT3.

5.1. JUNIT

Figure 4 shows the transaction overview of JUNIT in the time interval between December 3 2000 until October 18 2006. One can see easily that in the beginning of the project only *EG* [3] is checking in his changes to the repository.

Later, several different developers also committed their changes, namely *KB* and *CS*. In about the middle of the figure two very large distances between two consecutive transactions can be detected. Thus, in this time interval no check-ins have been done over a long period of time. The only active developer here seems to be *CS*. At the end of the analyzed time interval *DS* committed the largest transaction. The black marked files additionally show that many new files have been added at the end of the regarded time interval.

Figure 5 shows the file author matrix for JUNIT. The top-most part shows the root directory, the `doc` directory and the source-code directory of an old JUNIT version. The part below shows the directories containing the source of recent JUNIT versions as well as a directory with the name `tries` which has been used to test new ideas.

As the names of the authors are sorted, it is possible to recognize that *KB* is the developer with the highest number of check-ins. *DS* and *EG* are also very active. Except for these three, four additional developers have checked-in their changes into the JUNIT CVS repository. However, they have only performed a few changes as we can see regarding the only sporadic pixels in the matrix.

KB as well as *DS* and *EG* have changed a lot of different files. However, for *EG* it strikes out that he has done most changes of nearly all files in the old source directory, but only few changes at the source code of the recent version (except some tests and files in `samples`). Moreover, *EM*, who is the developer with the 4th most check-ins, has changed no file in the new source directory. Instead, *CS* has done some changes there.

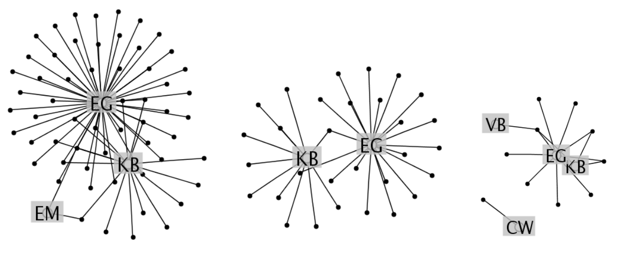
But, have the developers worked together (as team) on the files or does every developer work on his own part of the software system? Looking at the source code we find a lot of files that have been changed by all three (old source) respectively all four (new source) main developers. But it catches our eyes that in both source directories files and subdirectories exist which have been changed only by one or a part of the developer team. For the old source code we also see that nearly all files have been altered mainly by *EG*. The `tests` directory is the only one where *EG* and *KB* have done about the same number of changes. Here, both developers work on the same tests, but with a different amount of intensity.

[3]We are only using the initials of the developer names in this work.

In the new source directory, most files have been changed nearly only by *KB* or nearly only by *DS*. There are only few files on which both have worked with about the same amount—except the tests. One module that has been changed exclusively by *DS* is `request`, which is a submodule of `internal`. Furthermore, it is interesting that the "new" main developers *KB* and *DS* have done only little work at the documentation. Instead, the documentation has been mainly done by *EG*. However, the file `FAQ` has been checked-in mostly by *CW*. As this is also the only file that has been checked-in by *CW*, obviously *CW* is the owner of this important documentation file.

Figure 6 shows a small excerpt from the dynamic AFG. Within the selected period *KB* reduced the number of files that he worked on. In June 2002 he only committed changes to files that have also been changed by *EG*. The fact that all three graphs became smaller within these months might be an indication of a less dynamic development process.

In all three months *EG* contributed the most of the files. However, while he apparently concentrated on a different part of the software than *KB* in April 2002, *EG* covered most of the files changed in June 2002.



(a) March 2002 (b) April 2002 (c) June 2002

Figure 6. Three months of development on JUNIT. There were no activities in May 2002.

5.2. TOMCAT3

Figure 7 shows the transaction overview for TOMCAT3 in the time interval since October 19 1999 until November 21 2004. One typical aspect for this project is that in the first half of the analyzed time period many transactions have been done. The number of developers seems to decrease over the evolution of the project. At the end of the shown time interval only *HG* and *BB* have been working on TOMCAT3.

Figure 8 shows the complete file author matrix for TOMCAT3. For the viewer's comfort, we have manually annotated the module names in the matrix. Looking at the ma-

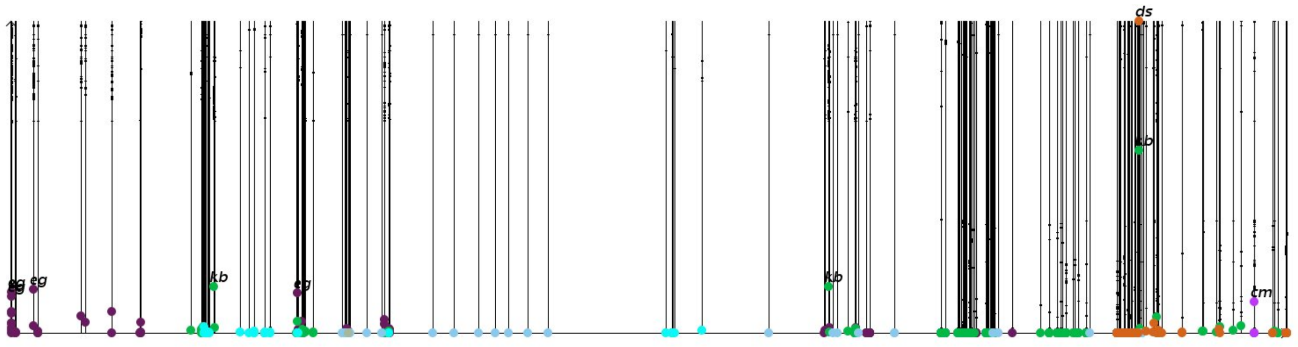


Figure 4. Overview of the transactions between Dec 03 2000 and Oct 18 2006 in JUNIT**.**

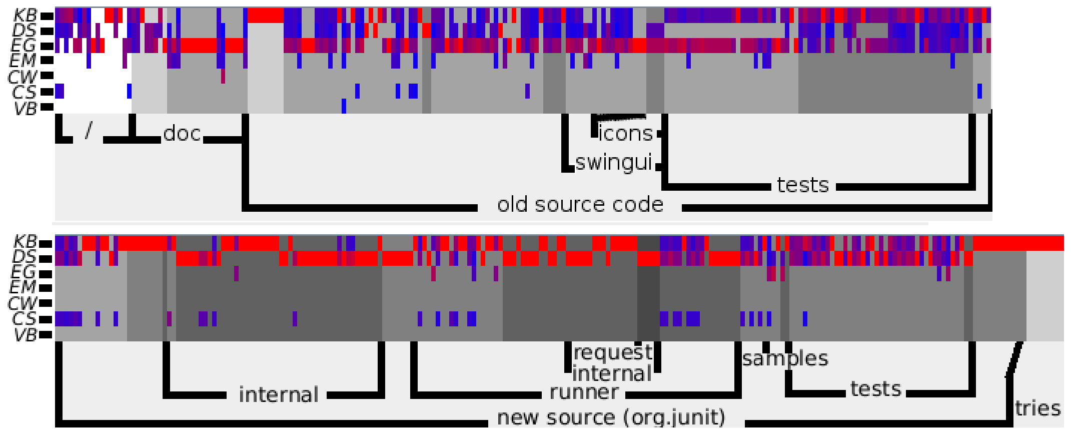


Figure 5. File Author Matrix of JUNIT

trix, we see that there are modules that have been changed by few (or even only one) developers (e.g., `j2ee`, `build`, and some modules within the tests), as well as modules that have been altered by 10 or more developers. The module that has been changed by the highest number of developers, is `org.apache.jasper`: 22 different developers have worked on this module, while for example the files within the module `j2ee` have been changed nearly only by *CM*, who has done the most check-ins of all developers.

Looking at the core of the TOMCAT3 project, i.e. the JAVA package `org.apache.tomcat` and its sub-packages, it caches our eyes that *CM* has contributed by far most of the changes. However, there are complete modules on which he has not worked much, among them the sub-modules of `src/native` which are programmed in C: these modules have been changed mainly by *BB*, *RS*, *GS* and *AL*. But also the module `org.apache.jasper` (see the previous paragraph) has not been changed by *CM*. Thus, we can say that *CM* is the main developer, but he leaves the work on single modules - mostly modules which are not part of the core - to other programmers.

We find it quite interestingly that the developer with the second most check-ins, *LI*, seems to have done quite little work at the source code. Instead, he has done a lot at the documentation within the `proposals` and at the web page. As this example shows, also people that do not work often on code, can be very important for a software project.

A look on the dynamic AFG for TOMCAT3 from September 2003 to February 2004 discovers a curios behavior of the two developers *HG* and *BB*. While *HG* contributed quite a lot in September and October he didn't commit any changes until end of February 2004. However, *BB* contributed only to a few files before he committed changes to nearly all files of the project. This observation is a indication for a change of the main developer role in this project.

Another interesting fact is the weak connection between the developers (except of February where *LI* performed only changes to files also changed by *BB*). TOMCAT3 seems to be developed in a separated process where each developer has his own domain of work.

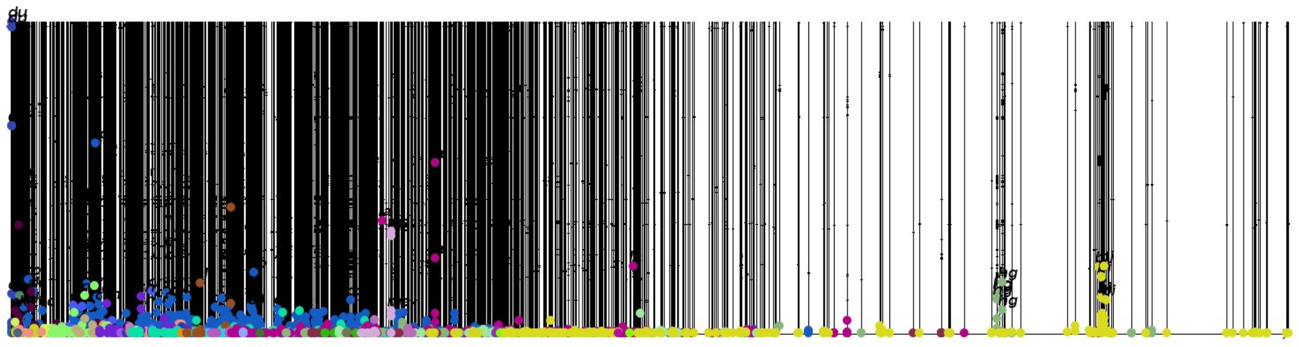


Figure 7. Overview of the transactions of TOMCAT3**, which have been made in the time interval since October 19 1999 until November 21 2004.**

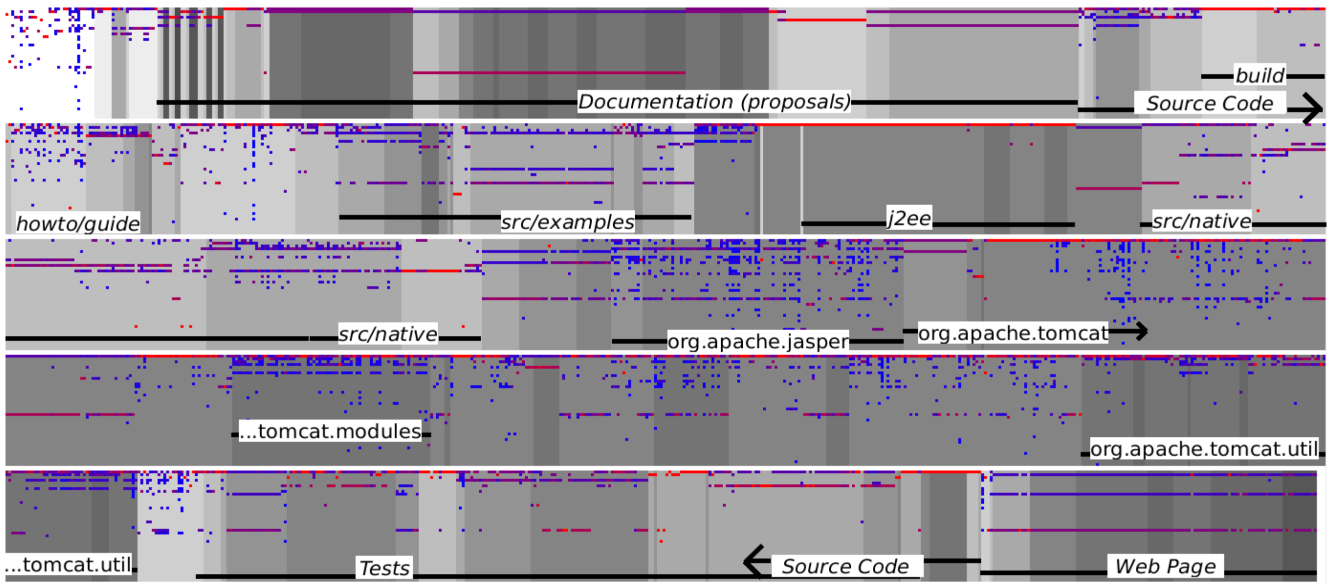


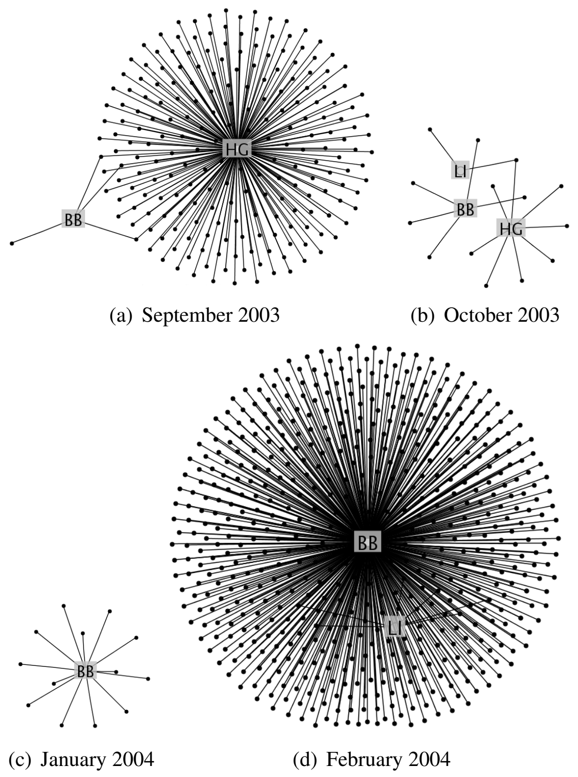
Figure 8. File Author Matrix of TOMCAT3

6. Additional Data Sources

In this work, we took only a look on quantitative changes in software archives. Beside the software archive, there are additional data sources that are interesting to examine for finding out how programmers work as a team:

- **Email archives:** Most projects have a mailing list on which the developers (which may be spread around the world) coordinate their work, and discuss ideas and recent changes. Thus, in email archives there is a lot of information on how developers work together. Recent work [3] on mining such archives also addresses challenges like alias detection (e.g., the same programmer uses different mail addresses).
- **Bug databases:** Bug tracking systems contain a functionality to comment each bug report. These comments are often used by developers to communicate how to solve exactly this particular issue. Thus, bug databases also contain information about which developers work together on which tasks. There are several approaches to combine bug database data and CVS data [8, 1] as well as to relate bug reports to source-code changes [13]. Recently, D'Ambros and Lanza proposed a combined visualization [6] for metrics retrieved from CVS data and from BUGZILLA to asses the evolution of a software project.

In future work, data of these additional sources should be merged with the CVS data and integrated into our analyses.



(a) September 2003 (b) October 2003

(c) January 2004 (d) February 2004

Figure 9. The AFG for TOMCAT3 **for four different months. There were no changes in November and December 2003.**

7. Related Work and Outlook

Apart from mining other data sources (see Section 6) most of the related work is from the area of software visualization and mining of software repositories.

Visualizing changes during the evolution of a software has been topic of recent research by several working groups. Using the SEESOFT tool [2] the changes of software artifacts over time can be examined. *Gevol* [5] helps to watch structural changes.

Other visualizations to show evolution of source code have recently been presented: CVSSCAN [15] by Voinea et. al. features a space-filling evolution overview for one file of a project. Each line of this file is represented by a pixel line on the screen. However, the line does not show the structure of the code (unlike as in SEESOFT). Instead the pixels in the line encode the subsequent versions of the file. The color of a pixel can encode several metrics, among them the author of the line (which enables to answer questions about how developers work together). However, while the evolution of each file can be examined on the granularity of lines, CVSSCAN can only show the evolution for one file at the same time, which makes it hard to get a global overview on all files of the project. This is indeed possible with our transaction overview (for each file it can be seen when which developer has done changes) and with the file author matrix (for each file it can be seen how many percent of the changes have been done by each developer).

Gîrba et. al. [12] came up with a similar space-filling visualization that, in contrast to CVSSCAN, works on the level of files and represents each file (the files are ordered hierarchically) by one line on the screen. Again, the pixels within the line are used to show the successive versions of the file. Each developer is mapped to a unique color and each pixel in the visualization is drawn in the color of the developer that owns most of the lines of that file in that version. Moreover, each change of a file is represented by a dot in the color of the developer. Using their visualization the authors were able to identify patterns in a project such as takeover (one developer performs so many changes in one session that he takes over ownership of a file), familiarization (one developer performs small changes on one or more files again and again until he gets ownership), bug-fixing periods and periods of silence. This visualization is very similar to our transaction overview, as it shows the evolution of all files over time. However, our visualization focuses more on showing the density of the transactions while Girba's technique nicely shows changes of code ownership.

The file author matrix visualization is related to the dependency structure matrix (DSM) [14]. We already used such visualizations to mark concurrent changes of files and methods [4] and to make statements about the module structure of a project [17].

Data mining techniques have been used by us [18] as well as by other researchers [9, 16, 10] to detect recurring development patterns. Such patterns can help to guide developers along related changes [18, 16] or to find starting points for reverse engineering [10]. However, these work focused more on the changes of the software itself than on the behavior of the developers.

Currently, we have independent tool prototypes for each of the three visualizations. As these visualizations work hand-in-hand we are planning to integrate these into one single tool that can be used e.g. by project managers for analyzing the social networks in their project.

8. Conclusion

In this paper we described by means of two case studies, how the combination of three visualization techniques helps to examine the development behavior of the programmers and the partitioning of the tasks between them in detail. The three visualizations work hand-in-hand: The transaction overview yields a global overview on all transactions that have been done during the project lifetime. Among other things, we can recognize when particular developers have been especially active. In the file author matrix one can see which developer(s) have worked together or alone on which files. Thus, using these two visualization techniques we can find interesting phases during the evolution as well as interesting relationships between the developers, which can be examined even more with the dynamic author file graph.

For both projects we have detected that there are multiple main developers. However, during the evolution of the projects the role of the main developer has been played by varying persons. While in JUNIT the main developers have performed changes in nearly all modules, we found for TOMCAT3 that there are also modules that have not been developed by any main developers. Moreover, we detected that in JUNIT many files have been altered by all main developers. In contrast, in TOMCAT3 there are such modules that are developed by one particular developer, as well as such ones that are altered by more than 20 developers. The development of JUNIT is divided into two major phases. Between these phases, there have only been relatively few transactions. For TOMCAT3 it cached our eyes that there was a lot of development from the beginning until the middle of the period under view. Later, the number of changes decreased tremendously.

The question, to what extent the programmers work as a team has to be answered for each project independently. In JUNIT the whole development has been concentrated on 1–2 main developers who have been supported in selective tasks by additional programmers. TOMCAT3 had a lot more active developers who partly worked together on some modules and partly worked independently on their own modules.

Acknowledgments

Patrick Reuther gave helpful comments on earlier revisions of this paper. Mathias Pohl is partially supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. DI 728/6-2.

References

[1] G. Antoniol, M. D. Penta, H. Gall, and M. Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electronic Notes in Theoretical Computer Science*, 127(3), 2005.

[2] T. Ball and S. Eick. Software Visualization in the Large. *IEEE Computer*, 29(4), 1996.

[3] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proc. of Int. Workshop on Mining Software Repositories MSR*, Shanghai, China, May 2006.

[4] M. Burch, S. Diehl, and P. Weißgerber. Visual Data Mining in Software Archives. In *Proc. ACM Symposium on Software Visualization SOFTVIS*, St. Louis, Missouri, USA, May 2005.

[5] C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A System for Graph-Based Visualization of the Evolution of Software. In *Proc. of ACM Symposium on Software Visualization SOFTVIS*, San Diego, USA, June 2003.

[6] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proc. of 10th European Conference on Software Maintenance and Reengineering CSMR*, Bari, Italy, March 2006.

[7] S. Diehl and C. Görg. Graphs, They Are Changing. In *Proc. of Int. Symposium on Graph Drawing GD*, Irvine, USA, August 2002.

[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. of Int. Conference on Software Maintenance ICSM*, Amsterdam, The Netherlands, September 2003.

[9] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proc. Int. Workshop on Principles of Software Evolution IWPSE*, Helsinki, Finland, September 2003.

[10] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proc. of Int. Conference on Software Maintenance ICSM*, Chicago, Illinois, USA, September 2004.

[11] C. Görg, P. Birke, M. Pohl, and S. Diehl. Dynamic Graph Drawing of Sequences of Orthogonal and Hierarchical Graphs. In *Proc. of Int. Symposium on Graph Drawing GD*, New York, USA, September 2004.

[12] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proc. Int. Workshop on Principles of Software Evolution IWPSE*, Lisbon, Portugal, September 2005.

[13] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr. Automatic identification of bug introducing changes. In *Proc. of Int. Conference on Automated Software Engineering ASE*, Tokyo, Japan, November 2006.

[14] D. V. Steward. The Design Structure System: A Method for Managing the Design of Complex Systems. *IEEE Transactions on Engineering Management*, 28, 1981.

[15] L. Voinea, A. Telea, and J. J. van Wijk. CVSScan: Visualization of code evolution. In *Proc. of ACM Symposium on Software Visualization SOFTVIS*, St. Louis, Missouri, USA, May 2005.

[16] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Transactions of Software Engineering*, 30(9), 2004.

[17] T. Zimmermann, S. Diehl, and A. Zeller. How History Justifies System Architecture (or Not). In *Proc. Int. Workshop on Principles of Software Evolution IWPSE*, Helsinki, Finland, September 2003.

[18] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions of Software Engineering*, 31(6), 2005.