Analytical Processing of Version Control Data: Towards a Process-Centric Viewpoint

Technical Report B-03-07

Dirk Draheim and Lukasz Pekacki
Institute of Computer Science
Free University Berlin
email: {draheim,pekacki}@inf.fu-berlin.de

May 2003

Abstract

This technical report introduces a novel approach to enabling analytical processing of project data. The approach exploits source code repositories for information about project evolution. Furthermore this technical report proposes a new perspective on analyzing version control data. It takes up a process-centric viewpoint, addresses related analysis problems like collaboration of programmers and proposes metrics for them. The research has yielded an implementation of the approach, which comprises visualizations that assist in examining the evolution of software process.

Contents

1 Introduction

Version control systems (VCS) contain large amounts of historical information that can give insight into the evolution of software projects. Unfortunately, they do not support analytical access to the project data.

This technical report contributes the following notions:

- *Analytical interface to version control data.* There is need for a clean cut abstraction layer for automatic access to VCS data. This need is satisfied by our implementation for the Concurrent Versioning System (*CVS*) [8]. It provides analytical access to the project data of most Open Source projects. It is designed for being used by researchers and practitioners for analyzing software projects.

- *Process-centric analysis of version control data.* It is desired to process VCS data from a software process-centric viewpoint. This particular viewpoint leads to exploration of process aspects that have been neglected in other studies. Visualization of process quality aspects points the way to precise modeling of process analysis problems - sound with respect to theory and practice of social research [1].

Most research in the field of software evolution focuses on the attributes of the software product itself [26][30][14]. Increasing knowledge about growth, complexity, refactoring or code decay has been gathered through the years. Yet, only few studies have examined aspects of process quality [31]. Software is coded by people and therefore its quality highly depends on the development practices that came into use. For comprehending this aspect of software evolution, empirical knowledge about process and human aspects is still needed [5].

This technical report presents a new infrastructure for accessing and analyzing project data. We provide an overview of our system in the next section. Afterwards, in section 3, we explain our change of perspective on project data towards a process-centric point of view. Examination of software development practices that are related to this new viewpoint follow in section 4. Finally, we give a brief overview of its implementation in section 5 and present tools that are built upon the system. The technical report closes with discussions on further and related work. The Appendix consists of example visualizations that demonstrate the capability of our software.

2 Analytical Interface for Project Data

In the course of the development of a software system, the current version of the software product is often the only up-to-date source of information about the state and the evolution of the system. The documentation itself is seldom synchronized with the versions and it becomes difficult to maintain a meaningful documentation of the system changes. But, current source code and documentation are not the only sources of data about the project. Presumably every software project uses some sort of VCS for keeping program files synchronized and consistent.

2.1 Concept

Project data, stored in a VCS, contains a huge amount of information about the evolution of the software product, the history of the development process and the individuals who contributed the code. A VCS enables users to get copies of the source code, and to add source code into the repository. It offers an interface for performing these file operations in a controlled way. During these operations the VCS stores additional contextual data that would be useful for analyzing the project. However, this data cannot be queried through the standard interface of the VCS. This gap can be bridged by introducing a new method for accessing the VCS.

Our concept is providing this interface that enables the user to analyze project data. The interface itself collects data through the default access method of the VCS. The interface is analytical because project data can be queried with high detail and complexity. It is flexible, because

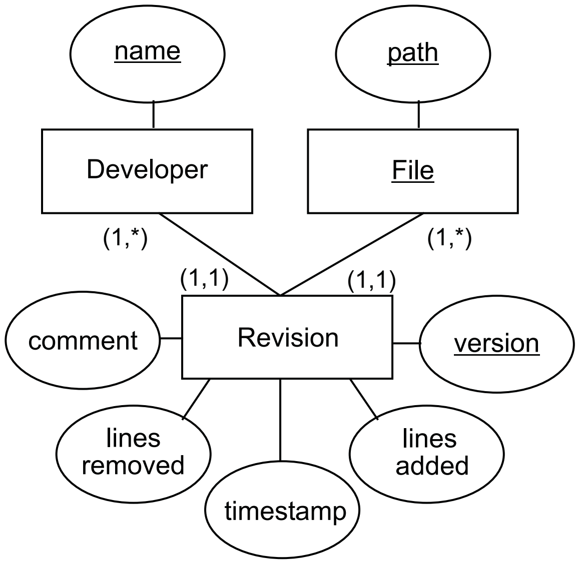

Figure 1: Data model of Bloof

generic requests can be performed on the data. In this technical report, we present its capability for analyzing process aspects of software projects.

2.2 Data Model

VCSs keep track of the changes to the files of a software system. In order to reconstruct past configurations of the source code most VCSs store the following context information for each change: the file name, the changed source code, number of added and deleted lines, a time stamp, a comment of the developer and a revision number that identifies the change. Some version control systems, like ClearGuide [24], store additional information like purpose of change, related change request, status of the associated request after the change etc. Although these information would also be useful for analysis, including them into the data model would require restriction to a specific VCS and would rule out many software projects from being analyzed. The data model is therefore reduced to a minimal version, making it compatible with virtually every version control system. In particular, it contains all data that is stored by the Concurrent Versioning System *CVS*, which is used throughout the Open Source community.

2.3 Benefits of Version Control Data

Collection and analysis of empirical software project data is central for improving software quality and programmer productivity. Unfortunately, collection and analysis of this data is rare in mainstream software development. Several barriers hinder data collection. The collection of empirical software engineering project data is expensive in resources and time, it often interferes with the work of developers and the validation of its accuracy is difficult. In addition, surveys on software projects tend to be designed for controlling development cost and seldom include useful data for developers or project managers. Overcoming these barriers in order to spread adoption of techniques of empirical software engineering is an important goal for future software engineering research. VCSs overbear some of these barriers.

VCS data collection is a side effect of the standard part of the development process and does not trouble developers. VCSs store long histories of running and past projects, facilitating analysis in the retrospective, even if no member of a project is available any more. The stored information is fine grained, down to the level of each atomic change. Collecting data on that level by hand is virtually impossible, especially in real life projects. Additionally, completeness of the data is assured automatically as every artifact under version control is recorded. The data is also uniform over time, as the way the version control system is used rarely changes. Because of the granularity of recording, even small projects generate enough data for analysis.

Although VCS data has these benefits, it is per se not designed for drawing conclusions about quality characteristics of a software project. In order to get this information - if it is possible at all - several assumptions on the data and further processing are necessary. These assumptions are discussed in section 4.

2.4 The *Bloof* Approach

Due to the high barriers for accessing project data, studies of software evolution tend to examine systems that have been developed in a single organization within a uniform, often traditional development and management environment [15][18][20][26]. Only recently, findings about Open Source Projects have been documented [19]. In order to reproduce and compare findings about software evolution, same empirical studies have to be applied to many different projects [16][20]. Significant results can only be provided by examining large sets of projects over long periods of time - not only on single snapshots [29][2].

A software system that addresses these needs should therefore provide easy access to the data of many software projects, include scalability to huge amounts of project data and support experimentation. In addition, it should be flexible in usage and provide interoperability.

These requirements are the design criteria for the *Bloof* system [32]. The system provides easy access to project data, because it accesses the data automatically. Speaking in terms of data warehousers, a version control system like *CVS* is an operative data source, whereas *Bloof* provides an ETL-layer (extraction, transformation, loading). *Bloof* is also scalable, because it transforms the data into a new data model and stores it in a database. It is also interoperable, as it provides a well documented analytical processing interface for performing data queries.

This design opens the way for various tools, examples of which are given in section 6. *Bloof* can be used for explorative analysis of project data, because it allows user defined queries. Finally, it can be used in a flexible way, because data access and data processing are implemented in a clean cut abstraction layer which can be integrated in various environments.

2.5 Usage Model

Although storing data in a database provides the possibility of querying, most of the interesting analysis problems cannot be queried in a simple database query statement. In order to ensure interoperability, a uniform result mechanism is also necessary. Therefore two ways of accessing project data information are supported:

- *Call level interface.* Generic SQL queries are supported by a query class with an SQL query string as only parameter.
- *Predefined analytical queries.* They are provided as compound queries, encapsulated in the query classes of the interface. They can be created with various parameters, and can run queries for complex analysis problems.

In both cases, a uniform return object is returned by the interface, which can be handed over to tools for further processing. This architecture allows different distributions of the *Bloof* system. A fat server distribution serves as web application for processing data of different software projects. Users trigger the loading of project data on the server and can perform queries on the fat server by using a web browser. A fat client distribution runs as local application and stores the loaded data in an internal database. A shell interface, a GUI or external tools accessing the interface of the *Bloof* system can use the same infrastructure. The current implementation of the *Bloof* system is coded in the *Java* language. Access to the analytical interface is realized as *Java API.*

3 Process Quality Centric Analysis of Software Project Data

Most of the studies of software evolution focus on examining the software product and its specific features. Little attention has been paid yet to the process that comes along with the evolution of the software. As software is mostly created manual - supported by various tools - the way how the programming has been accomplished by the developers determines to a high degree the properties of the result.

Product-oriented software evolution research concentrates on features like amount of source code, code decay, coupling, cohesion and their specific behavior over various releases. Many of these approaches make a good deal explaining the history of a software product and forecasting problems. They also are helpful for comprehending critical spots of a software product. Still, several aspects of the process have not been taken into account when solely concentrating on the product features.

Because software is to a great extend produced by hand, the resulting product highly depends on how people work and interact. Productivity, cooperation or continuity of work are main factors of process quality. We propose a new viewpoint on software projects, which addresses these features. Taking up a process-centric perspective we analyze the process of a software project. This approach is process-centric, because we analyze the activities of the developers during the process of software development.

4 Visualizing and Examining Software Development Practices

The evolution of a software system is usually complex, because it comprises the changes of large amounts of software artifacts and great numbers of developers over long periods of time. Visualization can help software engineers to cope with this complexity. The charts and tables that can be generated by the *Bloof* system facilitate comprehension of evolution through the display of product and process behavior, which otherwise is hidden in project data. The Appendix provides an assortment of example visualizations:

- Individual cumulative productivity measured in changed LOC per day.
- Team collaboration, comparing total changes with collaborative changes on a daily basis.
- Distribution of changes between main modules comparing the sum of added and deleted lines of code over the whole period of a project.
- Time line of changes per month.
- Average time since a file was changes the last time.
- Time line of hours that lay between the current change an the last change of a file.

In sections 4.1 and 4.2 we delve into two example analysis problems that are related to the process-centric perspective on software projects: productivity and collaboration. The analysis problems lead to the phrasing of queries on project data. The results of the related queries are shown in Figure 3 and Figure 4 in the Appendix. The visualisations has been generated by using version control data from the *GIMP* project [28]. The *GIMP* project is a good candidate for evolution research, because it is a heavyweight, highly successful Open Source project, used by many people, attracting many developers and being developed and maintained for more than 5 years, now. Having undergone many releases and changes of programming staff, it is a huge source of information about software evolution.

4.1 Examining Productivity

Basically, productivity is the rate of output per time period. In order to understand and manage productivity one has to measure it. However, measuring productivity in the context of software production is tricky and depends on many variables. Over decades, organizations implemented many different productivity metrics [10].

Commonly, productivity is measured by lines of code (LOC), although there are several drawbacks. The number of LOC used to implement functionality varies greatly between programming languages. Programmer productivity cannot be compared by measuring LOC if different programming languages are involved. Although this problem can be theoretically overcome by measuring function points instead of LOC, this solution has no practical use for automatic measurement, because function points can only be measured manually - by trained experts [21]. Different programming tasks also require different productivity measurement. Metrics for maintenance need to be kept separate from those for fresh development [11]. Project teams using different tools or working in different hardware/software environments also need to be measured and evaluated separately [3]. In addition, there is risk that programmers start working mainly for meeting the productivity metric variables - producing volume, not quality - or feel discouraged from programming reusable pieces of software [7][22].

It depends on specific features of the project and phase of production, which indicator provides useful information and which parameters should be considered for measuring LOC productivity. Although metrics are not perfect and are all subject to manipulation by programmers and management, used with care, they can help in identifying problems and risks, evaluating project decisions and predicting the future progress of the project. Various parameters for examining productivity can be fed into the *Bloof* system, enabling the user to configure queries according to her specific problem.

4.2 Examining Collaboration

Every process model for software development addresses the aspect of collaboration - not in the same way, though. A conventional software project might limit cooperation of developers to accepting tasks during meetings and defending results during reviews. In contrast, Extreme Programming fosters collaboration from the outset. Communication, one of the four values [4] of the method, is realized by a couple of best practices like pair programming and collective ownership.

Collaboration has obviously occurred when different people perform changes on same artifacts in a certain time frame. Artifacts can be modules of the system, directories, single files or file groups. Identifying these artifacts points to spots of a software product, where people have worked together. On the other side, taking up the viewpoint of a single programmer, her personal grade of collaboration can be identified by analyzing shared changes of artifacts. Putting both views together, groups of people can be identified, who are logically a team, although from the organizational point of view they might be not. Top collaborators can be found through ranking as can be those who work on their own. Adding the time perspective to these views, one gains insight into the evolution of the grade of cooperation. Any of these analyses of collaboration is supported by the *Bloof* system and can be performed as compound query.

However, it depends highly on the circumstances how to evaluate the results of these queries. The development process of the specific project and the personal work style of the programmers have to be taken into account when measuring collaboration. Attention also has to be paid to the interpretation of the results. In some cases, editing same files is not a good indicator for collaboration but one for bad division of work, bad design or poor communication. In fact, it is also possible that collaboration occurs, but cannot be tracked in the project data, since not everybody who performed changes also commits them to the repository, or - at least - does not commit them immediately. This is especially apparent in the case of pair programming environments where two people cooperate, but only one of them commits changes. Still, when considering these variables, collaboration metrics can help in evaluating the development process. The *Bloof* system supports the configuration of collaboration metrics by providing several parameters to a compound query.

4.3 Further Analysis Problems

Productivity and cooperation are not the only analysis problems on development process. Continuity and frequency of work, e.g., are also indicators for process quality which are implemented in the *Bloof* system. There also exist various questions which are supported by our software that do not directly address a process quality problem but still answer interesting questions related to the process quality, like e.g.

- A developer changes only files that she created herself.
- A developer only deletes lines of code.
- A developer submits only small changes.
- A developer works on a large amount of files.

These observations can also be queried in the negative from, e.g. "A developer works on a small amount of files". Performing these queries in a ranking can identify high potentials. Adding the dimension of time gives insight into past evolution and provides indicators for prediction. Adding a third dimension, e.g. "Productivity time line based on sub modules of the system" allows deep analysis of the project history. These observations can all be queried on the analytical interface of *Bloof*.

Although this technical report focuses on process-centric analyses, the implementation is not limited to this focus. Product-centric analyses like those conducted by various research studies mentioned in section 8 are also supported by the system and are implemented as standard queries, too.

5 Implementation for *CVS*

Bloof is designed to provide easy access to the data of many software projects, to allow interoperation with other tools and to support experimentation on project data. *Bloof* is an Open Source project being hosted by *Sourceforge*. It supports the version control system *CVS*. *CVS* is not only used by many commercial organizations and research institutes, but also by most of the Open Source projects, especially by over 60.000 projects which are located at *Sourceforge*. Interoperation with other tools is realized in the *Bloof* software by separating the data access and analysis layer from the application layer as shown in Figure 2. The *Java* implementation allows the usage of the system on various platforms. Internally there is a hierarchy of query classes all of which return a unified result object, either as Java object or as XML document. Detailed information about *Bloof* is available on the project website [32].

6 Tools for Project Data Analysis

The architecture of *Bloof* allows access to analysis of VCS data via the *Java API*. Several tools that use this interface are developed in the *Bloof* project. The main distribution of the *Bloof* system includes a GUI tool, the *Bloof Browser*, which enables the user to perform data access, analysis and visualization. Data artifacts can be navigated, filtered and grouped. The tool provides a set of compound queries, visualizes the results and enables the user to export them into a XML document. A shell for querying the data model via *SQL*, a web server based distribution and external visualization tools are also included.

7 Further Work

Data sources like *Sourceforge* open up the way for processing data of large numbers of projects. Comparing the results of many projects could reveal patterns of software development. One of the

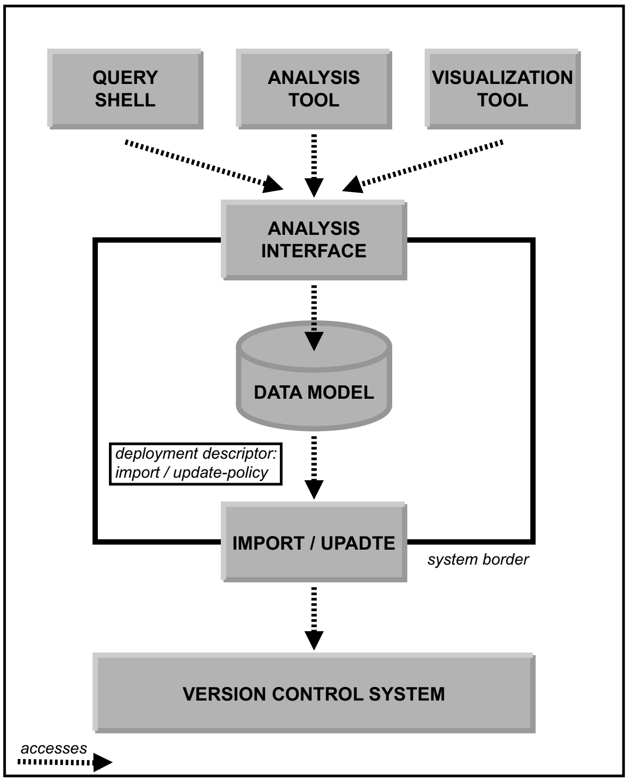

Figure 2: Architecture of the *Bloof* system

tasks to be done in the future of the *Bloof* system is the implementation of a meta project layer. This work would also provide access to cross-project analysis of developer activities. This might be appealing not only for researchers but for project managers who are interested in benchmarking [6], too.

The data model can be extended by external information. More specific data about atomic changes can be imported from systems like ClearGuide. Adding information about events in the development process, like dates of releases, would allow automatic analysis of patterns of evolution during or short after events. Enriching the data with more information about a change, e.g. the actual changed content or line number, would allow more fine-grained analysis of the source code.

Including statistical tests into the calculation of the results can lead to strong predication of correlation and significance of variables.

8 Related Work

Long term research on large software systems, which has been conducted by Lehman et al., produced a large source of information about evolution of software products [27][26] and led to the phrasing of several laws of software evolution. Results of this work included models for growth and complexity, prediction of errors, and guidelines used by project managers in planning software development [25].

Evolution aspects of the software product have been investigated by many other researchers. Analyzing frequency of changes, grow rates and change types contributed to comprehending the evolution of software and finding reasons for success or failure in a software project [19].

Fine grained analysis produced visualization of structural changes during evolution of software systems and has introduced several change patterns of software artifacts [23].

Expertise of developers, measured on VCS data, has been recently researched by Mockus [31].

The approach related developer expertise to developer activity on artifacts over the time. It was measured automatically and without preconditions on the project. In some aspects, this work is similar to our approach, however, it analyzes only one single aspect of process and does not address the problem of data access.

The combined CSCW/project management tool PEASE [13] (platform for EASE) already offers limited statistic features for analyzing collaboration. However the statistic features of PEASE can only be used reasonably in projects that are managed strictly with the special-purpose process model EASE [12]. Moreover the statistic features rely on explicit data about collaboration that is stored in a special project data repository from the outset. No analytical interface is provided and the analysis results are pure reports, i.e. they are not visualized.

Yet, surveys of research about software evolution note that there has been relatively little research on empirical studies of software evolution [19].

Suggestions for exploiting version control data for analyzing software evolution were explored in the Bell Labs [2], some years ago. The metrics and visualizations provided there where focused on aspects of the software product, like e.g. logical coupling of files, and did not take examinations of different versions of the product or changes over time periods into account.

Software evolution studied, which use version control data, do often not provide information about accessing the data. A step in this direction has recently been performed by transforming version control data and bug tracking data into a database schema, allowing simple queries on the project data [17]. Like many other studies before, this work relies on manual data access and transformation. It does not aim to integrate provision of automatic data accessing, analytical processing and result generation into one system.

Targeting this goal, the Open Source project *StatCvs* [9] released a product which generates a static suite of web pages, filled with charts and tables, which contain metric results about the history of a software project. Although *StatCvs* made it to interest many users, lacking of scalability, flexibility and interoperability in the design of *StatCvs* led to the creation of the *Bloof* system.

9 Conclusion

The *Bloof* system is suitable for analyzing software projects which have their source code under version control. Even small projects provide sufficient data for analytical processing. This solution removes the barrier of access to software project data. The system is easy to integrate due to its lightweight interface. It enables flexible analyses of evolution aspects, because it provides a simple query-result mechanism and supports complex data queries. It is particularly capable of visualizing process aspects of evolution, as the analyses from a process-centric viewpoint show and also supports more prevalent examinations of evolution aspects of a software product. The system satisfies the demand for a testbed on software evolution and the *Bloof* tools satisfy the requirements for tools for software evolution. They enable its users to perform explorative experimentations on project data. *Bloof* aims to provide an infrastructure for empirical research of software evolution.

References

[1] E. R. Babbie. *The Practice of Social Research, 10th ed.* Wadsworth, 2004.

[2] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk. In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[3] R. Banker, R. Kaufman, and R. Kumar. An empirical test of object-based output measurement metrics in a computer aided software engineering (case) environment. *Journal of Management Information Systems*, 8(3):127–150, 1992.

[4] K. Beck. *Extreme Programming Explained - Embrace Change.* Addison-Wesley, 2000.

[5] K. Bennet and V. Rajlich. *The Future of Software Engineering*, chapter Software Maintenance and Evolution: a Roadmap. ACM Press, 2000.

[6] C. E. Bogan and M. J. English. *Benchmarking for Best Practices: Winning Through Innovative Adaptation.* McGraw-Hill Trade, 1994.

[7] C. Byard. Software beans: Class metrics and the mismeasure of software. *Journal of Object Oriented Programming*, 7(5):32–34, 1994.
[8] P. Cederqvist. Version management with CVS. http://www.cvshome.org/docs/manual/. 1992.
[9] R. Cyganiak, A. Jentzsch, L. Pekacki, and M. Schulze. Statcvs - stat your repository. http://statcvs.sourceforge.net/, 2002.
[10] C. Dale and H. van der Zee. Software productivity metrics: Who needs them? *Information and Software Technology*, 34(11):731–738, 1992.
[11] D. Davis. Does your IS shop measure up? *Datamation*, Sept 1:27–32, 1992.
[12] D. Draheim. Learning software engineering with EASE. In T. J. van Weert and R. K. Munro, editors, *Informatics and the Digital Society*, pages 119–128. Kluwer Academic Publishers, 2003.
[13] D. Draheim. A CSCW and project management tool for learning software engineering. In *Frontiers in Education - Engineering as a Human Endeavor*. IEEE Press, to appear.
[14] S. Ducasse, M. Lanza, and L. S. Software. Supporting evolution recovery: A query-based approach. Software Composition Group, University of Berne, 2000.
[15] S. G. Eick. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 6(1):1–12, 2001.
[16] N. Fenton and S. L. Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2 edition, 1996.
[17] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. Technical Report TUV-1841-2003-06, Information Systems Institute, Distributed Systems Group, Technical University of Vienna, 2003.
[18] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings: 1997 International Conference on Software Maintenance*, pages 160–166. IEEE Computer Society Press, 1997.
[19] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM*, pages 131–142, 2000.
[20] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July/Aug. 1999.
[21] W. Keuffel. Metrics conclusions. *Software Development*, June:29–32, 1995.
[22] R. Kliem and I. Ludin. Making reuse a reality. *Software Development*, 3(12):63–69, 1995.
[23] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002*, pages 135–149, 2002.
[24] D. B. Leblang. Managing the software development process with ClearGuide. In *Software configuration management: ICSE 97 SCM-7 Workshop*, pages 66–80. Lecture Notes in Computer Science 1235, Springer, May 1997.
[25] M. Lehman. Feast/2 final report. http://www.doc.ic.ac.uk/ mml/feast. 2001.
[26] M. Lehman, D. Perry, J. Ramil, W. Turski, and P. Wernick. Metrics and laws of software evolution-the nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics'97)*, Albuquerque, NM, 1997.
[27] M. M. Lehman. *Program Evolution: Processes of Software Change*, chapter 12, pages 247–274. Academic Press, London, UK, 1985.
[28] P. Mattis and S. Kimball. Gimp - gnu image manipulation program. http://www.gimp.org/, 2003.
[29] T. Mens and S. Demeyer. Evolution metrics. In *Proc. Int. Workshop on Principles of Software Evolution*, 2001.
[30] A. Mockus, S. Eick, T. Graves, and A. Karr. On measurement and analysis of software changes. Technical report, National Institute of Statistical Sciences, Research Triangle Park, NC, 1999.
[31] A. Mockus and J. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *ICSE '02 Workshop on Open Source Software Engineering*, Orlando, FL, USA, 2002.
[32] L. Pekacki. Bloof - visualize software project evolution. http://bloof.sourceforge.net/, 2003.

Appendix

This Appendix consists of figures showing data that has been imported from various *Sourceforge* projects and processed by the *Bloof* system.

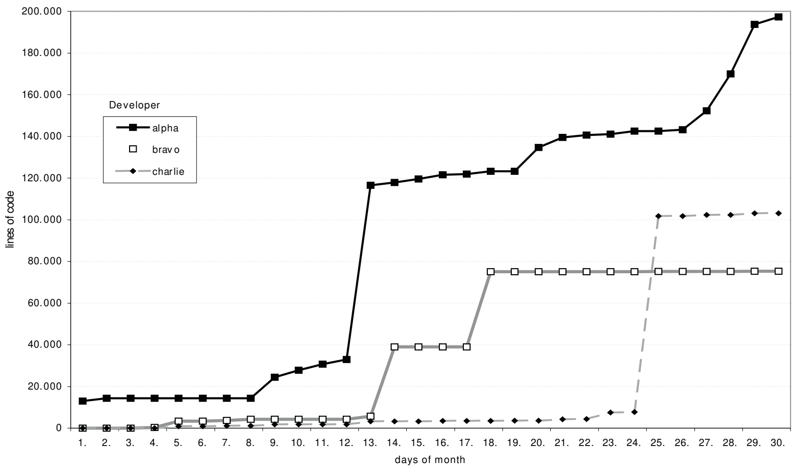

Figure 3: Cumulative productivity measured in changed LOC per day, generated by the *Bloof* system (*GIMP* project, processed for Nov. 2001, comparing three developers)

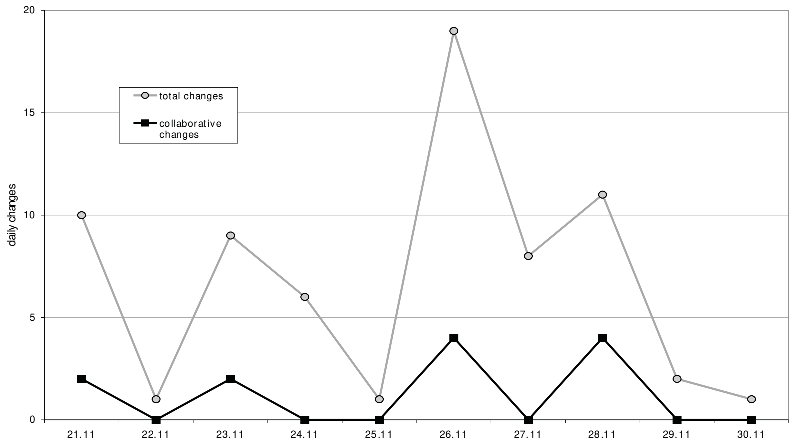

Figure 4: Time line of collaboration, comparing total changes with collaborative changes on a daily basis (*GIMP* project, processed for the time period of the final 10 days of Nov. 2000, summarizing contribution of all developers)

Module	**%**
swingui	33%
framework	17%
tests	13%
textui	13%
runner	11%
awtui	8%
other	5%

Figure 5: Distribution of changes between main modules in the *Junit* project, comparing the sum of added and deleted lines of code over the whole period of the project.

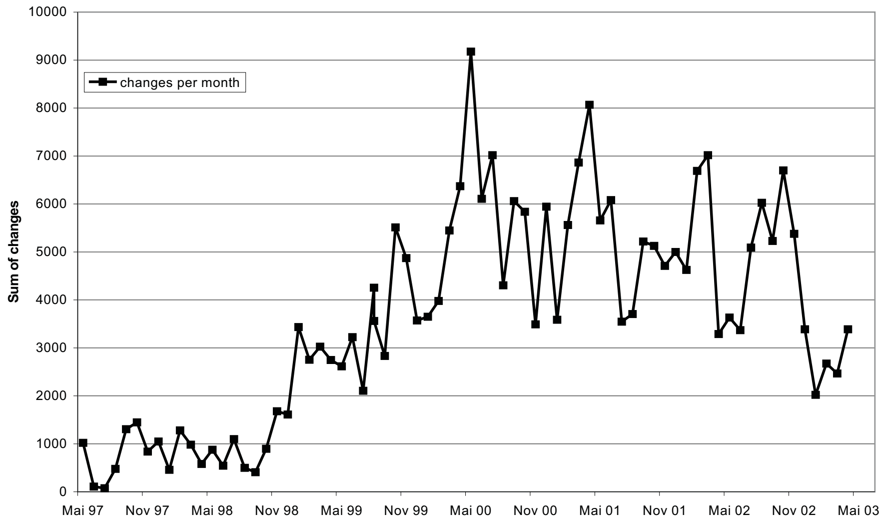

Figure 6: Time line of changes per month. (*KDE* project)

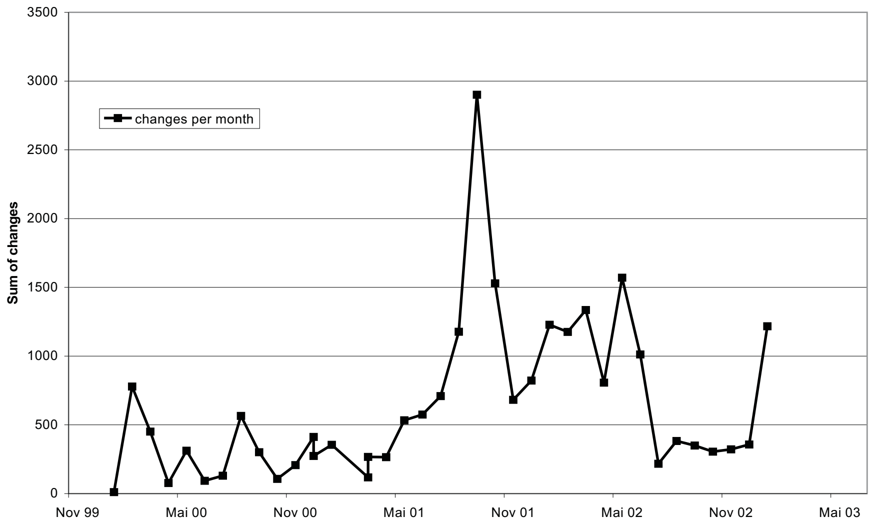

Figure 7: Time line of changes per month. (*JEDIT* project)

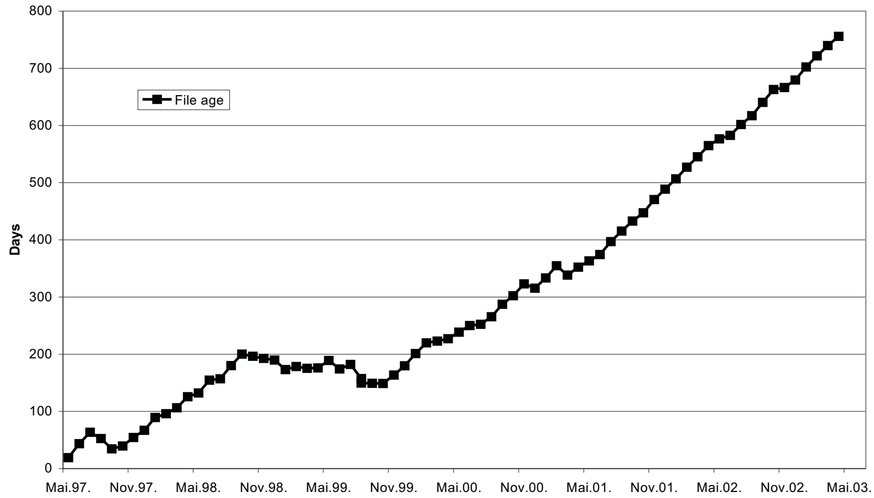

Figure 8: Average time since a file was changed the last time. (*KDE* project)

.

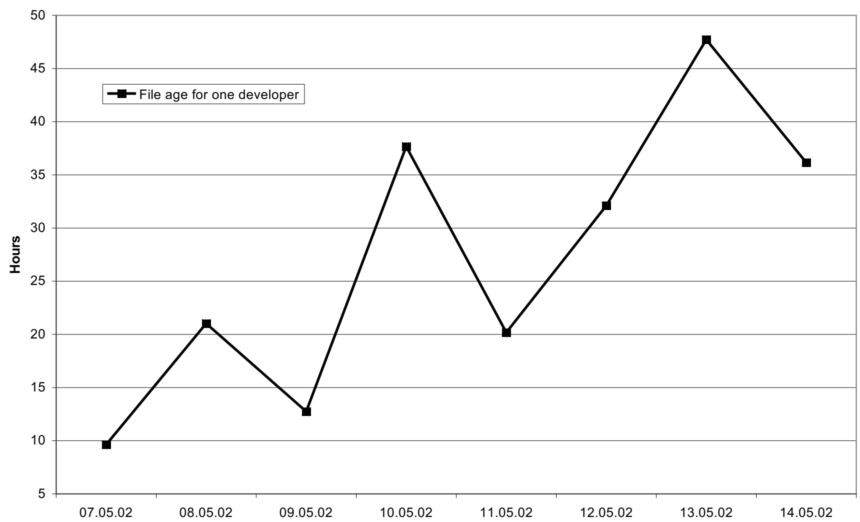

Figure 9: Time line of hours that lay between the current change an the last change of a file. (*JEDIT* project, processed for one developer for the time period of a week in May 2002)