

# A Framework for Describing and Understanding Mining Tools in Software Development

Daniel M. German

Davor Čubranić

Margaret-Anne D. Storey

Software Engineering Group, Dept. of Computer Science  
University of Victoria, Box 3055 STN CSC, Victoria BC  
Canada V8W 3P6

{dmg, mstorey, cubranic}@uvic.ca

## ABSTRACT

We propose a framework for describing, comparing and understanding tools for the mining of software repositories. The fundamental premise of this framework is that mining should be done by considering the specific needs of the users and the tasks to be supported by the mined information. First, different types of users have distinct needs, and these needs should be taken into account by tool designers. Second, the data sources available, and mined, will determine if those needs can be satisfied. Our framework is based upon three main principles: the type of user, the objective of the user, and the mined information. This framework has the following purposes: to help tool designers in the understanding and comparison of different tools, to assist users in the assessment of a potential tool; and to identify new research areas. We use this framework to describe several mining tools and to suggest future research directions.

## 1. INTRODUCTION

Understanding how programs evolve or how they continue to change is a key requirement before undertaking any task in software engineering or software maintenance. Software engineering is a highly collaborative activity and hence *awareness* is an important factor in being informed of what has changed and what is currently being changed.

Software teams consist of many different stakeholders with distinct roles in their projects. A developer is interested in knowing how related artifacts changed in the past and why these changes occurred. A reengineer wants to consider how a system has evolved so that they can learn from prior experiences before redesigning the system. A manager is interested in understanding ongoing development and a programmer's previous work before assigning new work. A researcher wants to study how large projects have evolved so that the lessons learned can be applied to new projects. And a tester wishes to know which parts of the program to test, and who to talk to if they have questions or problems to report. Some of the many questions these various stakeholders ask of a software project can often be answered by other team members. In some

cases the relevant team members may no longer be available or they may not remember important details adequately. Therefore, answering these questions requires the extraction of information from a project's history to answer a particular stakeholder's questions. Unfortunately, these questions often do not have a simple answer. Details concerning concrete changes can be extracted from a source code repository, but the intent behind these changes is not easy to infer without considering other information sources and doing some sort of deeper analysis.

During the past few years, many researchers have started to investigate how software repositories and other information sources can be mined to help answer interesting questions which will inform software engineering projects and processes. Most of these research projects originate from trying to solve particular problems that satisfy different user needs.

In a recent paper [15], we presented a framework to describe how awareness tools in software development use visual techniques to present relevant information to different stakeholders. We used this framework to provide a survey of visualization tools that provide awareness of human activities in software engineering. The framework considered the intent behind these tools, their presentation and interaction style, the information they presented, as well as preliminary information on their effectiveness.

We noted in this earlier survey that the visualization tools are limited in their effectiveness by the information available to display. For example, if a tool only extracts information about software releases, the tool will not be able to reveal who made the changes, no matter how sophisticated the visualization technique may be. Extracting information from most information sources is relatively straightforward. But many questions can only be answered by correlating information from multiple sources. The difficulty of successfully mining pertinent information arises during this analysis. It is challenging to know which questions to ask and how best to answer the questions given that some of the information may be incomplete or vague. An example is relating an email message to a particular change in the source code when trying to discover intent. Another problem is that such information repositories although rich, are often very large and contain many details that are not relevant to the problem at hand. It is also important to know how to filter the information so that the user is not overwhelmed by a deluge of data.

The goal of this paper, therefore, is to complement our visualization framework by exploring and analyzing the issues related to the mining aspects of software tools. In our previous work we stud-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, Saint Louis, Missouri, USA Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

ied the issues related to the presentation of information to the user, while in this paper we focus on the data available and its extraction. A framework for mining software repositories should enable us and other researchers to understand how these diverse mining tools are positioned within a broader research context. It should provide a mechanism for tool researchers and designers to evaluate and compare their work with other efforts, as well as illuminate new research areas which could benefit software engineering.

In the first part of this paper, we summarize the different user roles and the specific tasks that can be supported by mining software repositories. We then explore, in depth, the different types of information that can be beneficial to these user roles while considering what kinds of analyses are needed to discover pertinent information. Finally, we demonstrate the benefits of this framework by comparing three diverse research tools that were independently developed by the three authors. Each of these three tools mines or extracts information from software repositories to support software engineering tasks. The framework helps us understand how these tools may be improved and highlights the need for more analysis of combined information sources.

## 2. A FRAMEWORK FOR COMPARISON

The framework for comparing software visualization tools of human activities is described in detail in [15]. Here we focus on just three of its dimensions, where each attempts to describe a different aspect of a repository mining tool. “Intent” explains who are the expected users of the tool, and its main objective. “Information” describes the specific sources that the tool mines and the type of analysis made by the tool. This dimension is elaborated in more detail as it is most relevant to mining software repositories. We provide some examples of tools to strengthen the descriptions of information extraction where necessary. Finally, the “infrastructure” addresses any special needs that the tool has.

### 2.1 Intent

We describe this dimension in detail in our other paper [15], but summarize it here.

**Role.** This dimension identifies who will use the tool. Roles include *developers*, and whether they are a part of a *team* that is co-located or distributed. Other development roles include *maintainers*, *reverse engineers* and *reengineers*. *Managers*, *testers* and *documenters* can also improve their effectiveness by knowing about human activities in the project. And finally *researchers* may wish to explore human activities to make recommendations for improved tools and processes on future projects.

**Time.** Some tools provide information about activities occurring in the distant or near *past*, while other tools focus on presenting information about the *present*. Other tools try to forecast the *future* and predict which parts of the system are more prone to be modified in the future.

**Cognitive support.** Cognitive support describes how a tool can help improve human cognition [16]. In order to provide cognitive support, it is essential to know which tasks require extra tool support. Specifically we need to know which questions are likely to be asked during these tasks and how the questions can be answered. The questions that the various roles can ask about developer activities can be roughly classified into four categories: *authorship*, *rationale*, *chronology*, and *artifacts*. Consequently, we consider

how mining tools can provide information according to these four categories.

### 2.2 Information

As we mentioned previously, this dimension is thoroughly explored as it is the most relevant to mining software repositories. To help clarify the discussion when necessary, we give specific examples of tools.

**Change management.** *Configuration management* tools provide support for building systems by selecting specific versions of software artifacts [7]. *Version control* tools contribute to software projects in the following ways: software artifact management, change management and team work support [18]. Change management is an important data source because it provides *traceability*: it records who performed a given change, and when it was performed. The capabilities of the change management system will determine the type of information that can be extracted. For instance, CVS does not record when a given commit is a branch-merge and it does not support transactional commits. Several heuristics have been created to overcome these problems [4, 5].

**Program code.** We classify these tools into two categories. In the first category we place those tools that treat the file as a unit, and make no effort to understand its contents; we call these tools **programming-language-agnostic**. On the other hand, tools are **programming-language-aware** if they attempt to do some fact extraction from the source code. We can further classify programming-language-aware tools based upon:

- **The language supported.** Given the differences in syntax and grammar, tools that are language-specific can only understand a fixed set of programming languages.
- **Syntactic analysis.** In this type of analysis the extractor does not need to understand what the code does, only its syntax. Examples of this analysis are the removal of comments from the source code (to be able to distinguish if the changes affected actual source code or only its documentation), and extraction of the main entities of the code (such as packages, classes, methods, functions, etc.).
- **Semantic analysis.** This analysis requires an understanding of the intent of the source code and can be done *dynamically* (by running the software under well defined test-cases) or *statically* (by processing the source code). The generation of a call graph, or the tracing of the execution of a program are examples of this type of analysis.

**Defect tracking.** Many larger software projects rely on tracking tools to help with the management of *defects* and *change requests*. Such systems often store metadata about who is assigned a task and track the task’s completion. In some cases a defect management tool is also used as a way to track activities and changes in requirements. For example, Bugzilla includes a category for a defect report called “improvement”, which is used by its users to submit a change in requirements.

**Correlated information.** We have observed that the type of analysis and correlation can be classified into two broad categories:

- **Within the data source.** This type of analysis uses data from one data source only and attempts to correlate different data

entities within it. In some cases this analysis strives to reconstruct relationships that were lost because they were not explicitly recorded (such as grouping file revisions into commits in CVS). In other cases, the new information is computed from the data available in the source (for example, extracting the functions that were modified in a given change). Some sources are very rich in the amount of information that can be extracted and correlated from them (version control systems are one example).

- **Between the data sources.** In some cases, there is explicit information that allows a tool to correlate entities from two different data sources. For example, it is not uncommon for open source developers to record the corresponding Bugzilla defect number in the log of the CVS commit that resolves such defect, allowing a tool to correlate file revisions with a defect. Frequently, there is no explicit information that correlates information from different sources, and heuristics are required to build these relationships. For example, which email messages are relevant to a particular bug fix?

**Informal communication.** Email is undoubtedly the most widely used form of computer-mediated communication, and it is not surprising that distributed software development projects rely on it extensively. In the early days of open-source software, a project mailing list used to be one of the first, and often the only, communication and coordination mechanism used by development teams [2]. Today, specialized tools like Bugzilla have taken over some of its functionality, but email remains an essential component of distributed development process. For example, open-source projects typically document all decisions on the project mailing list, even when the original decision was reached in a different medium, or such as face-to-face [8].

In recognition of the mailing list's importance to a project, it is usually archived and available on the web. However, messages in the archive are typically organized chronologically or at best by conversation thread. Even when text search of an archive is available, finding specific information can be difficult. For example, if a developer wants to know why certain a function was added to the project, then the challenge is to find all the messages that relate to the decision to add that function. The limited structure and metadata of archived email mean that this source of information is rarely mined. However, in their study of developer communication in open-source projects, Gutwin et al. found that developers would like to see improved access to email archives [8].

Various forms of text chat, such as IRC and IM, have become increasingly important channels of communication in open-source projects. For example, in 2000, neither Apache nor Mozilla projects had official IRC channels used by the development team, and today both do. Text chat is rarely archived (and when it is, it is usually in another form, such as email messages, or as part of a Web page), but this is likely to change as its importance is recognized. However, chat has even less structure than email, so it may be considerably more difficult to mine effectively.

Advances in computing technology are making it possible to archive communication that used to be unarchivable. For example, Richter et al. have demonstrated a system for automated capture of team meetings [11]. Their system provides automated transcript of the spoken content, which the attendees can annotate on-the-fly with a set of keywords from a predefined list.

**Local history.** Many local interactions are not captured by a project's repositories. However, a developer's local history is a rich resource for understanding human activities and how they relate to the software under development. Recently, several researchers have been investigating how mining this information source can assist in navigation and program comprehension.

Two tools that address navigation support are Mylar and NavTracks. Mylar [9] provides a degree-of-interest model for the Eclipse software development environment. As a program artifact is selected, its value increases while the value of other artifacts decrease. Therefore, elements of more recent interest have a higher degree of interest value. Mylar filters artifacts from the Package Explorer in Eclipse that are below a certain threshold and thus helps a developer focus on the artifacts in the workspace that are relevant to the current activity. Navtracks [14] is a tool to support browsing through software spaces. It provides recommendations of files that should be of higher relevance to the user given the currently selected file. It keeps track of the navigation history of a software developer, forming associations between related files. Associations are created when short cycles are detected between file navigation steps. There are also several projects in the human interaction research community that investigate how tracking interaction histories can support future interactions [17, 1].

Schneider et al. describe how local interaction histories can be mined to support team awareness [13]. They propose that sharing local interactions among team members can benefit the following activities: coordinating team member activities such as undo, identifying refactoring patterns and coordinating refactoring operations, mining browsing patterns to identify expertise, and project management. They describe a tool called Project Watcher and are currently evaluating the benefits it brings to developers.

## 2.3 Infrastructure

This category addresses the environment needed to support the tool.

**Required infrastructure.** This category lists any requirement the tools have, such as a given operating system, an IDE such as Eclipse, a Web server and client, a database management system, etc.

**Offline/Online.** Tools can be classified depending upon whether the software repository is required during its operation. For instance, some tools mine the software repository ahead of time, while others query the repository as a result of a user request.

**Storage backend.** If the tool operates offline, this category is used to describe how it stores its required data. For example, some tools use a SQL backend, other use XML or a proprietary format.

## 3. A COMPARISON OF MINING

We now use this framework to help us understand the intent and mining capabilities of three tools designed by the authors.

### 3.1 softChange

**Intent:** The main goal of softChange is to help programmers, their managers and software evolution researchers in understanding how a software product has evolved since its conception [6]. With respect to *time*, softChange concentrates only on the past. In terms of *cognitive support*, it allows one to query who made a given change to a software project (*authorship*), when (*chronology*) and, whenever available, the reason for the change (*rationale*). The *artifacts*

that softChange tracks are files, and some types of entities in the source code (such as functions, classes, and methods).

**Information:** softChange extracts and correlates three main sources of information: the version control system (CVS), the defect tracking system (Bugzilla), and the software releases. softChange reconstructs some of the information that is never recorded by CVS (such as recreating commits), and it does syntactic analysis of the source code. The analysis is static and it supports C/C++ and Java. softChange also attempts to correlate information between CVS and Bugzilla using defect numbers.

**Infrastructure:** softChange is an offline tool that uses an SQL database for its storage needs. Its mining is done without any special requirements beyond access to the software repository. One particular problem with the type of mining that softChange does is that it can retrieve a very large amount of data, and for that reason, it is recommended that it operate on a local copy of the repositories (rather than query the repositories using the Internet, consuming their bandwidth and computer resources). softChange has two different front ends: one is Web based, and the other a Java application.

### 3.2 Hipikat

**Intent:** Hipikat can be viewed as a recommender system for software developers that draws its recommendations from a project's development history [3]. The tool is in particular intended to help newcomers to a software project. Therefore, in terms of the *time dimension*, it is concentrated on the past. *Cognitive support* is largely limited to answering questions about *rationale* and *artifacts*. In terms of user *roles*, Hipikat is targeted almost exclusively at developers and maintainers.

**Information:** Hipikat is designed to draw on as many information sources as possible and identify relationships between documents both of same and different types. The information sources that are currently supported in Hipikat are: version control system (CVS), issue tracking system (Bugzilla), newsgroups and archives of mailing lists, and the project Web site. All four of these sources are typically present in large open-source software projects.

Hipikat is programming language-agnostic. The only information that it collects from files in the version control system is versioning data, such as author, time of creation, and check-in comment.

Hipikat correlates information across sources using a set of heuristics, such as matching for bug id in version check-in comment to link file revisions in CVS and bug reports in Bugzilla. These heuristics are based on observations of development practices in open-source projects like Mozilla. Another method that Hipikat uses to find documents that are related is by textual similarity.

**Infrastructure:** Repository mining in Hipikat works in offline mode: Hipikat periodically checks project repositories for recent changes and updates its model. The model is stored in an SQL database. The front end is an Eclipse plug-in, although in principle it could be implemented for other environments, as long as it follows the communication protocol with the Hipikat server.

### 3.3 Xia/Creole

**Intent:** The main goal of the Xia [18] tool is to help *developers* understand version control activities by visualizing architectural dif-

ferences between two versions. Therefore, within the *time dimension*, it focuses on the past, both near and distant. Xia provides *cognitive support* for developers when they need answers to questions concerning *authorship*, *chronology*, and *artifacts*. Several of the visual techniques from Xia have been subsequently integrated into the Creole visualization plug-in for Eclipse [10]. The purpose of the Creole tool is to provide both high-level visualizations of the architecture of a program as well as detailed views of dependencies between software artifacts. Combining views from Xia with Creole means that information concerning version control activities can be viewed in concert with the dependency views in Creole.

**Information:** Creole represents software using a graph where nodes in the graph correspond to software artifacts such as packages, classes, methods, fields, etc., and edges correspond to relationships such as "created by," "calls," and "accesses data". Creole extracts information from the CVS version control system and tags nodes in the graph with the following information: authorship (author of first commit, last commit, and the author with the most number of commits); time (time of first commit and most recent commit) and the total number of commits. This information can then be used in tooltips for the artifacts in the repository, or to filter nodes from the view or to highlight them using a colour scale.

**Infrastructure:** Creole and Xia both work in *online* mode and directly access the CVS repositories. Creole extracts dependencies from the source code using the Feat data extractor [12]. For large projects, CVS queries can be very slow. Creole and Xia have both been integrated with Eclipse as plugins. Creole is available for download from [www.thechiselgroup.org/creole](http://www.thechiselgroup.org/creole).

## 4. DISCUSSION AND CONCLUSIONS

Tools need to be created around the needs of the developer. To our knowledge, very little has been done in terms of asking developers what types of requirements they have, and few tools have been formally evaluated to determine if they are useful to their expected users. Many of the tools are created around the needs of the researcher (somebody who is interested in understanding how a software system has evolved). This is a natural phenomenon because many of these tools are built by researchers to satisfy their own requirements. We could argue that by coincidence some of the requirements of the managers are the same as those of the researcher. Developers, however, have a different type of questions that need answers. Researchers and managers are frequently satisfied with trends and aggregated data; the developer, on the other hand, requires precise answers most of the time. Once the needs of the potential users are better understood (the **intent**), then one can determine what information should be mined and how it can be analyzed (the **information**).

Some data sources are very rich, and others have been barely exploited. The more data retrieved, the more difficult it will be to find relevant information for a given query (high recall) with little noise (high precision). One can argue that the act of "mining" is not the important problem that tools are trying to solve. Instead, these tools are attempting to answer valid questions that their users have by taking advantage of the historical information available. Tapping into new sources of data should be done with relevance in mind. How can this data be used to help answer a question? Who is the potential user? What questions can it help answer?

The less structured or organized the historical information is, the

more difficult it is to use it effectively. We conjecture that the reason why few tool use email messages (and other informal forms of communication) is because they are difficult to correlate to other types of information, and to answer questions posed by the user. However, the informal forms of communication are being recorded, and in the future, they could prove to be an important source of valuable information.

We hope that this paper prompts discussion towards a common nomenclature, and potentially, an ontology that can be used to describe tools that mine software repositories. Another area that we believe should be considered is the selection of a set of applications that can serve as test cases or benchmarks (this has been already suggested during the previous Workshop in Mining Software Repositories in 2004). It would then be possible to create a corpus with copies of the software repositories, that can be shared among the researchers; this will reduce the stress posed to the servers of the projects that are to be mined.

Having a common set of benchmarks will also help to address another problem in the area of mining software repositories. The actual task of retrieving “facts” from the repository is not considered to be an important research issue. Often, the act of mining involves reverse engineering of the formats in which the data is stored, scraping information from the Web, or trying to find some regularity in the output of tools that access the repositories. In this case (such as the syntactic and semantic analysis of source code) it involves the use of tools created by other communities (such as the program analysis and comprehension communities); sometimes the problem is getting the tools to work with the information retrieved from a particular repository. The act of mining for facts is tedious and error-prone. If the community agrees on a set of test cases, the fact extraction can be done only once, and the resulting data shared along with the copies of the repositories. This will allow researchers more time to concentrate on the more important problems related to the analysis and, correlation of this information always keeping in mind the needs of the potential user.

## 5. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments.

## 6. REFERENCES

- [1] M. Chalmers, K. Rodden, and D. Brodbeck. The order of things: Activity-centred information access. In *Proceedings of 7th Intl. Conf. on the World Wide Web (WWW7)*, 1998.
- [2] D. Čubranić and K. S. Booth. Coordinating open-source software development. In *Eighth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 61–65, 1999.
- [3] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Learning from project history: A case study for software development. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 82–91, 2004.
- [4] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32. IEEE Computer Society Press, September 2003.
- [5] D. M. German. Mining CVS repositories, the softChange experience. In *1st International Workshop on Mining Software Repositories*, 2004.
- [6] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softChange. In *Proc. of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, pages 336–341, 2004.
- [7] J. C. Grundy. Software architecture modeling, analysis and implementation with SoftArch. In *the Proceedings of the 25th Hawaii International Conference on System Sciences*, page 9051, 2001.
- [8] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *Proc. of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81, 2004.
- [9] M. Kersten and G. Murphy. Mylar: A degree-of-interest model for IDEs. In *Proceedings of Aspect Oriented Software Development*, March 2005.
- [10] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. In *Proc. of the 2003 ACM Symposium on Software Visualization*, pages 47–56, 2003.
- [11] H. Richter, G. D. Abowd, C. Miller, and H. Funk. Tagging knowledge acquisition to facilitate knowledge traceability. *International Journal on Software Engineering and Knowledge Engineering*, 14(1):3–19, Feb. 2004.
- [12] M. Robillard and G. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of 25th International Conference on Software Engineering*, May 2003.
- [13] K. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer’s local interaction history. In *Proceedings of 1st International Workshop on Mining Software Repositories*, 2004.
- [14] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software space. In *International Workshop on Program Comprehension*, 2005. To be presented.
- [15] M.-A. Storey, D. Čubranić, and D. M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2nd ACM Symposium on Software Visualization*, 2005. To be presented.
- [16] A. Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In *Proc. of the 11th International Workshop on Program Comprehension (IWPC’03)*, pages 185–195, 2003.
- [17] A. Wexelblat. Communities through time: Using history for social navigation. In T. Ishida, editor, *Lecture Notes in Computer Science*, volume 1519, pages 281–298. Springer Verlag, 1998.
- [18] X. Wu, A. Murray, M.-A. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proc. 11th Working Conference on Reverse Engineering*, pages 90–99, 2004.