MSR 2006 Organization

General Chairs: Stephan Diehl *(University Trier, Germany)*
Harald Gall *(University of Zurich, Switzerland)*
Ahmed E. Hassan *(Research in Motion RIM, Canada)*

MSR Challenge Chair: Martin Pinzger *(University of Zurich, Switzerland)*

Program Committee: Premkumar T. Devanbu *(University of California, Davis, USA)*
Daniel German *(University of Victoria, Canada)*
Mike Godfrey *(University of Waterloo, Canada)*
Ric Holt *(University of Waterloo, Canada)*
Shih-Kun Huang *(National Chiao Tung University, Taiwan)*
Jane Huffman Hayes *(University of Kentucky, USA)*
Katsuro Inoue *(Osaka University, Japan)*
Michele Lanza *(University Lugano, Switzerland)*
Tim Lethbridge *(Ottawa University, Canada)*
Jonathan Maletic (*Kent State University, USA)*
Ken-ichi Matsumoto *(NAIST, Japan)*
Audris Mockus *(Avaya Labs, USA)*
Leon Moonen *(Delft University of Technology, The Netherlands)*
Thomas J. Ostrand *(AT&T Labs, USA)*
Dewayne Perry *(University of Texas, USA)*
Jelber Sayyad Shirabad *(Ottawa University, Canada)*
Alexandru Telea *(Eindhoven University of Technology, The Netherlands)*
Kenny Wong *(University of Alberta, Canada)*
Annie Ying *(IBM Research, USA)*
Thomas Zimmermann *(Saarland University, Germany)*

External Reviewers: Marco D'Ambros, Cathal Boogerd, Michael L. Collard,
Natalia Dragan, Huzefa Kagdi, Mircea Lungu, Masao Ohira,
Romain Robbes, Andrew Sutton, Peter Weißgerber,
Jingwei Wu, Shehnaaz Yusuf, Liming Zhao

Introduction to MSR 2006

Stephan Diehl
FB IV – Informatik
Universität Trier
Trier, Germany
diehl@acm.org

Harald Gall
Martin Pinzger
Department of Informatics
University of Zürich
Zürich, Switzerland
{gall,pinzger}@ifi.unizh.ch

Ahmed E. Hassan
Performance Engineering
Research In Motion (RIM)
Waterloo, Canada
ahmed@alumni.uwaterloo.ca

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Miscellaneous

General Terms

Algorithms, Management, Measurement

ABSTRACT

Software repositories such as source control systems, defect tracking systems, or archived communications between project personnel are used to help manage the progress of software projects. Software practitioners and researchers are beginning to recognize the potential benefit of mining this information to support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques. Research is now proceeding to uncover the ways in which mining these repositories can help to understand software development, to support predictions about software development, and to plan various aspects of software projects.

Following the success of the first two iterations of the MSR workshop in 2004 and 2005, MSR 2006 attracted even more submissions: We received 45 papers from 15 different countries. The international program committee accepted 16 full and 12 short papers for presentation at the workshop and inclusion in the proceedings. We are grateful for the excellent and professional review job done by the reviewers on such a tight schedule.

1. GOAL AND TOPICS

The goal of this two-day workshop is to establish a community of researchers and practitioners who are working to recover and use the data stored in software repositories for further understanding of software development practices. We expect the presentations and discussions in this workshop to continue on a number of general themes and challenges, from the previous MSR workshops held at ICSE 2004 and 2005, a recent TSE special issue on the MSR topic, and the Dagstuhl-Seminar on Multi-Version Program Analysis held in Summer 2005. The workshop covers themes and topics such as:

- Engineering tasks related to the infrastructure and tools needed to recover useful data from repositories
- Methods of integrating mined data from various data sources
- Development and validation of approaches to visualize and present such data
- Use of recovered history for system understanding and analysis of change patterns
- Models of defects and software reliability using data from such repositories
- Uncovering of the social processes and interaction between the development community
- Discovery of techniques to facilitate software reuse

As the field of mining software repositories has become more mature, we expect MSR 2006 to be a forum for exploratory work as well as continuing work. In 2006 we want to foster systematic comparisons of different approaches in our field. To this end, MSR 2006 includes a challenge session in addition to the demo session.

2. MSR 2006 CHALLENGE

The MSR Mining Challenge brings together researchers and practitioners who are interested in applying, comparing, and challenging their mining tools and approaches on software repositories. The this year's challenge covers the two well known open source software projects PostgreSQL and ArgoUML. 12 mining reports address the development process, team structure, change coupling, bug resolution, and cross-cutting concerns. 4 reports concentrated on analyzing ArgoUML, 5 on PostgreSQL, and 3 on analyzing both projects. The results of all 12 reports present valuable insights into both open source projects; for instance, did you know that the main contributors to PostgreSQL comprise only two people?

The reports and the program of the MSR Mining Challenge are the results of hard work. First of all, we would like to thank the authors of submitted reports. Many thanks goes to the open source community and in particular to the

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

ArgoUML and PostgreSQL project teams for sharing their project data. They enable us to develop, compare, and challenge our mining approaches and tools.

We further would like to thank Beat Fluri, Patrick Knab, and Sandro Boccuzzo for helping with the reviews, and Martin Pinzger, Harald Gall, Michele Lanza, and Marco D'Ambros for organizing the MSR challenge. We are looking forward to the participation in the final round of the MSR Mining Challenge to see the best mining tools.

For more information on MSR and the MSR Challenge we refer to `http://msr.uwaterloo.ca/`.

Mining Large Software Compilations over Time: Another Perspective of Software Evolution*

Gregorio Robles, Jesus M. Gonzalez-Barahona
Universidad Rey Juan Carlos
{grex,jgb}@gsyc.escet.urjc.es

Martin Michlmayr
University of Cambridge
martin@michlmayr.org

Juan Jose Amor
Universidad Rey Juan Carlos
jjamor@gsyc.escet.urjc.es

ABSTRACT

With the success of libre (free, open source) software, a new type of software compilation has become increasingly common. Such compilations, often referred to as 'distributions', group hundreds, if not thousands, of software applications and libraries written by independent parties into an integrated system. Software compilations raise a number of questions that have not been targeted so far by software evolution, which usually focuses on the evolution of single applications. Undoubtedly, the challenges that software compilations face differ from those found in single software applications. Nevertheless, it can be assumed that both, the evolution of applications and that of software compilations, have similarities and dependencies.

In this sense, we identify a dichotomy, common to that in economics, of software evolution in the small (micro-evolution) and in the large (macro-evolution). The goal of this paper is to study the evolution of a large software compilation, mining the publicly available repository of a well-known Linux distribution, Debian. We will therefore investigate changes related to hundreds of millions of lines of code over seven years. The aspects that will be covered in this paper are size (in terms of number of packages and of number of lines of code), use of programming languages, maintenance of packages and file sizes.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement, languages

Keywords

Mining software repositories, large software collections, software evolution, software integrators

*The work of Gregorio Robles, Jesus M. Gonzalez-Barahona and Juan Jose Amor has been funded in part by the European Commission under the CALIBRE CA, IST program, contract number 004337. The work of Martin Michlmayr has been funded in part by Google, Intel and the EPSRC. We would also like to thank the anonymous reviewers for their extensive comments.

MSR'06, May 22–23, 2006, Shanghai, China.

1. INTRODUCTION

Large systems based on libre software[1] are developed in a manner that is quite different to traditional systems. In traditional large systems, such as operating systems, most work is done in-house, with only few pieces licensed from other sources and little work contracted to other companies. Such work is also performed in close cooperation with the organization and under tightly defined requirements. Libre software, on the other hand, is typically written by small, independent teams of volunteers, sometimes collaborating with paid staff from one or more companies. While various projects interact with each other, in particular where dependencies between the software exist, there is no central coordination between the individual projects. The main task of vendors (i.e. distributions) of libre operating systems is therefore not to write software but to group existing software, taken from several sources, together and to make that collection easy to install, configure and administer.

Since users of libre software have no incentive to download software from hundreds of sites and installing them individually, distributions play an important role by providing an integrated system that is easy to install. Unsurprisingly, a number of companies have seen this as a business opportunity and offer such distributions among with related services, such as support. There are also a number of community projects which operate on a non-profit basis like other libre software projects. Given their open way of collaboration, these are a good target for in-depth study of extremely large software compilations. While some commercial entities have recently started their own community projects in addition to their enterprise offerings, most notably Fedora (Red Hat) and OpenSUSE (Novell), we will take Debian as the source

[1]Through this paper we will use the term "libre software" to refer to any code that conforms either to the definition of "free software" (according to the Free Software Foundation) or "open source software" (according to the Open Source Initiative).

of data for this study since it is one of the most accessible and best established projects.

Debian is a community effort that has provided a software distribution based on the Linux kernel for well over 10 years. The work of the members of the Debian project is similar to that carried out in other distributions: software integration. Unlike many other distributions, Debian is mostly composed of volunteers who are spread all around the world. As a side-effect of this, all development infrastructure, including mailing lists, bug tracking and of course the source code itself, is publicly available. In addition to integrating and maintaining software packages, members of the Debian project are in charge of the maintenance of a number of services, such as a web site, user support, etc. In the following, we will mostly focus on the work carried out in their role as integrators of software – work that has had tremendous success, given that Debian is the largest distribution of all in terms of number of software packages [1].

2. RELATED RESEARCH AND GOALS

Software evolution has been a matter of study for more than thirty years now [5, 7]. So far, the scope of software evolution analyses has always been that of single applications. Example case studies are the "classical" analysis of the OS/360 operating system [5], and, more recently, many of studies on libre software systems. Such is the case for the Linux kernel [3], or other well-known libre software applications, including Apache and GCC [11]. Noteworthy is the proposal of studying the evolution of applications at the subsystem level [2], as this introduces the issue of granularity. Nonetheless, our approach considers as system the whole software compilation and as subsystem the hundreds of applications and libraries that are usually matter of software evolution studies.

However, the authors have not found a study on the evolution of a system integrating many independent software applications. Actually, software compilations have rarely been studied in software engineering. This is probably due to the intrinsic difficulties that software companies find when integrating large amounts of software programs built by several vendors. There are a number of reasons for this, both legal and technical. It seems that even if one of the most promising steps of software engineering has been to create reusable components (or modules), in a similar way as bricks and mortar, little attention has been put on how the integration of these components evolve. A promising path has been the study of integration of COTS from a software evolution perspective [6].

As noted above, the public availability of source code of libre software programs and the possibility of freely redistributing this software allow to have an ample number of software distributions. Both characteristics also enable the investigation of distributions. In this sense, there have been already some *radiographies* of some distributions, mainly of the well-known Red Hat and Debian distributions. These studies have pointed out the packages they contain, the size of the packages and of the whole distribution, and some statistics on the programming languages, among other issues [14, 4, 1].

This paper goes a step beyond the *single-version* analyses of software distributions: our goal is to study the evolution of software compilations. We therefore consider data from several points in time. However, it should be noted that the goals of this study differ slightly from those usually considered as common for software evolution. In part this is because a different type of work has to be accomplished while creating software compilations than during software development. The work to be done for a software compilation is mainly integration of software rather than development, although the latter is not excluded at all (for instance, for the development of an installer or other software administration tasks that distributions may include). Needless to say, there are some aspects that are common to *traditional* software evolution analyses, such as how the size of the software evolves.

Putting a software distribution together is not only integration work, however. Maintenance also has to be performed, but not so much in the *classical* way as defined by Swanson (corrective, adaptive and perfective maintenance activities) [12]. Maintenance in software compilations focuses on the integration of new versions of software that has been released. In other words, a package maintainer will not necessarily submit patches that correct errors; but they will update the package whenever new versions are published by the developers of the application or when changes in the distribution, such as library transitions or toolchain updates, occur. This raises interesting questions in our longitudinal analysis. For instance, we will analyze packages that are kept and that get *lost* (removed) over time, as the composition of the software compilation may vary. We will also look at packages whose version has not changed, as we will take this as an indication of *unmaintained* packages.

As software compilations are composed of a large variety of software applications for different purposes and from different backgrounds, we may find a larger heterogeneity than when looking at specific software applications. This is the case for instance in the use of programming languages: a particular software application, for example the Linux kernel [3, 10], is usually implemented primarily in one programming language, with only minor portions in other languages (such as glue code or the build system). This means that studying compilations as large as the one we have selected as our case study can be considered as a proxy of libre software in general – a macroscopic view of the libre software landscape. We are in this sense performing a holistic study of libre software and analyze how it is *in the large*, drawing some conclusions about the phenomenon itself.

3. METHODOLOGY

The methodology that we have used for the analysis of the stable versions of Debian is as follows: first, we have retrieved files which contain information about the packages that are distributed in a given Debian distribution. Distributions are organized internally in packages where packages correspond to applications or libraries. Debian developers commonly try to modularize packages to the maximum, for example splitting documentation into a separate packages if it is very large. Since 2.0, the Debian repository contains a Sources.gz file for each release, listing information about every source package. For each package, it contains the name and version, list of binary packages built from it, name and e-mail address of the maintainer, and some other information that is not relevant for this study. In some cases, packages are not maintained by individual volunteers, but by teams.

As an example, an excerpt of the entry for the Mozilla

source package in Debian 2.2 has been included below[2]. It can be seen how it corresponds to version M18-3, provides four binary packages, and is maintained by Frank Belew.

```
[...]
Package: mozilla
Binary: mozilla, mozilla-dev, libnspr4, libnspr4-dev
Version: M18-3
Priority: optional
Section: web
Maintainer: Frank Belew (Myth) <frb@debian.org>
Architecture: any
Directory: dists/potato/main/source/web
Files:
  57ee230[...]c66908a 719 mozilla_M18-3.dsc
  5329346[...]bad03c8 28642415 mozilla_M18.orig.tar.gz
  3adf83d[...]ca20372 18277 mozilla_M18-3.diff.gz
[...]
```

The Sources.gz files are parsed and the data they contain is stored into a database. Then, each package is retrieved to a local machine, the number of source lines of code (SLOC) is counted and the programming languages in which the code is written are recognized. The counting is made by means of SLOCCount[3], a tool written by David Wheeler that gives the number of physical source lines of code of a software program. SLOCCount takes as input a directory where the sources are stored, identifies (by a series of heuristics) the files that contain source code, recognizes for each of them (also by means of heuristics) the programming language, and finally counts the number of source lines of code they contain. SLOCs are parsed differently for different languages, which forces the identification of programming languages.

SLOCCount also identifies identical files (by using MD5 hashes), and includes heuristics to detect (and avoid counting) automatically generated code. These mechanisms are helpful when analyzing the code, but have some deficiencies. Finding almost identical files using such hashes is not very effective. In the second case, heuristics only take care of well-known and/or common cases, but do not detect all of them, or others that may appear in future. Nevertheless, SLOCCount is a proven tool and it has has been used on studies on Red Hat [14] and on Debian [4].

The results of the SLOCCount analysis are transformed afterward into other formats, including both relational and XML data formats. Hence, with a simple web interface anyone can have access to raw data and more elaborated visualization forms that facilitate a first analysis (graphs, maps, among others). Many of the results carried out for this study are offered in a web site[4].

4. RESULTS AND OBSERVATIONS

In the following subsections we are going to present and discuss the results obtained from applying our methodology to several Debian releases.

4.1 Observations on the size of Debian

At the time of publication, the latest stable release of Debian is version 3.1, also known under the codename *sarge*. The testing version has been codenamed *etch* and will become the next stable Debian version some time in the future. Finally, the one that is in development is called *sid*. In the past, *sarge* also passed through this testing phase. What we are going to consider in this work are the stable versions of Debian since version 2.0, published in 1998. Thus, we will consider Debian 2.0 (*hamm*), Debian 2.1 (*slink*), Debian 2.2 (*potato*), Debian 3.0 (*woody*) and, finally, Debian 3.1 (*sarge*). The codenames of the versions in Debian correspond to the main characters of the animated cartoon film *Toy Story*.

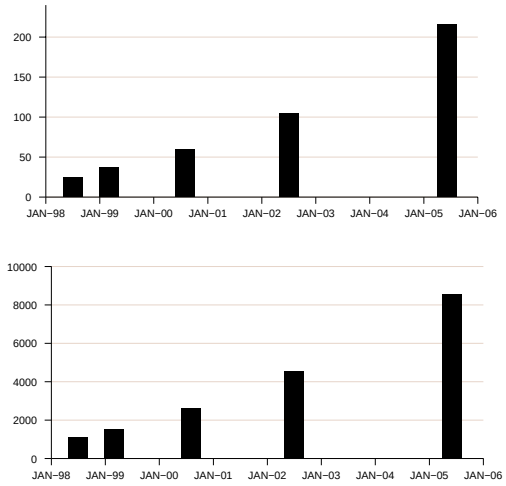


Figure 1: Size in MSLOC and number of packages for the versions in study. Top: MSLOC for each version. Bottom: Number of packages for each version. In both cases, the studied versions are spaced in time along the X axis according to their release date.

In figure 1 the number of MSLOC and source packages for the considered stable versions of Debian can be found. Debian 2.0, released July 1998, includes 1,096 source packages that have more than 25 MSLOC. The following stable version of Debian, version 2.1 (published around nine months later), contains more than 37 MSLOC in 1,551 source packages. Debian 2.2 (released 15 months after Debian 2.1) sums up around 59 MSLOC in 2,611 packages, whereas the next stable version, Debian 3.0 (published two years after Debian 2.2), groups 4,579 packages of source code with almost 105 MSLOC. Finally, almost three years later, Debian 3.1 has been released, with 8,560 source packages and more than 216 MSLOC.

Version	Release	Source pkgs	Size	Mean pkg size
2.0	Jul 1998	1,096	25	23,050
2.1	Mar 1999	1,551	37	23,910
2.2	Aug 2000	2,611	59	22,650
3.0	Jul 2002	4,579	105	22,860
3.1	Jun 2005	8,560	216	25,212

Table 1: Size of the Debian distributions under study. Size is given in MSLOC, while the mean package size is in SLOC.

[2]The original Sources.gz file where this entry comes from can be found at http://www.debian.org/mirror/list.
[3]http://www.dwheeler.com/sloccount/
[4]http://libresoft.dat.escet.urjc.es/debian-counting/

Although the number of points is not sufficient to make an accurate model, we can infer from the current data that the Debian distribution doubles its size (in terms of source lines of code and of number of packages) around every two years, although this growth has been much more significant at the beginnings (from July 1998 to August 2000 we observed an increase of 135%) than in later releases (between July 2002 and June 2005 the source code base has not achieved a 100% increase even though 3 years have passed). Hence, using time in the horizontal axis, we would have a smooth growth of the software compilation as found by Turski [13]. On the other hand, if we considered only releases (which is the methodology preferred by Lehman), the growth would be super-linear basically because the time interval between subsequent releases has been growing for most recent releases.

4.2 Observations on the size of packages

The histograms in figure 2 display package sizes for Debian 2.0 and Debian 3.0 (measured in SLOC). It can be clearly observed that large packages grow in size with time, while at the same time more packages near the origin appear. It is astonishing how many packages are *very small* packages (less than thousand lines of code), *small* (less than ten thousand lines) and *medium-sized* (between ten thousand and fifty thousand lines of code).

A small number of large packages in size (over 100 KSLOC) exist and the size of these packages tends to increase over time, as the sixth *law* of software evolution states [8]. Nevertheless, it seems surprising that in spite of the growth that Debian has undergone, the graph does not show big variations. Still more interesting is the fact that the mean size for the packages included in Debian is slightly regular (around 23,000 SLOC for Debian 2.0, 2.1, 2.2, 3.0 and 3.1, see table 1). With the data available at the present time it is difficult to give a solid explanation of this fact, but we can suggest some possible hypotheses[5]. As packages tend to grow in size and if no new packages are added to new versions of Debian, a growth in the mean package size would be expected. So it is the inclusion of new, small packages that makes the mean size stay almost constant for around seven years. Perhaps the *ecosystem* in Debian is so rich that while many packages grow in size, smaller ones are included causing that the average to stay approximately constant.

4.3 Observations on the maintenance of packages

Up to the moment, we have seen how Debian has been growing in the last 7 years as far as the number of packages and the number of SLOC is concerned. In the following paragraphs, we will attend an opposite dimension: packages that have not changed. This has to be understood in the sense that taking care of a software distribution requires maintaining packages, i.e. among other activities including new versions of the packages in the distribution. Packages that maintain from one release to the other the same version number may have been maintained actively, but usually even performing corrective maintenance implies releasing new versions of the software. We can therefore assume that no changes have been performed if the version number has not been changed.

It should be noted that Debian has a policy about version numbers. In addition to the software version, the project appends an own revision number. So, for example, a package with version number 1.2-3, means that it is the third Debian revision (upload) of that package in its 1.2 version. So, even if the original software is not maintained, the Debian versions may change because of library transitions (for instance, compiler changes from GCC 3 to GCC 4). Thus, some packages have Debian revision numbers up to 20 or higher - simply because the original software is not developed anymore but Debian still maintains that package.

Figure 4.3 will help explaining how we are going to measure maintenance activity supposing that we have two distributions (given each one by a set of packages, in the figure these are Debian 2.0 and Debian 3.1). The circle that gives the set of packages for the Debian 3.1 version has a larger radius as it contains many more packages than Debian 2.0 (the area of the circles could be considered as proportional to their size in number of packages). Both sets may have packages in common (the intersection between the two sets, as it is the case for the kernel-source package). Other packages will only be included in one of them. If packages appear only in the older Debian version, we say that it has been *lost*, while packages that appear only in the newer one are new (or added) packages. We can also identify a subset of those packages that remain with the same version number (a subset of the intersection between the two sets); those are the packages that we will consider unmaintained.

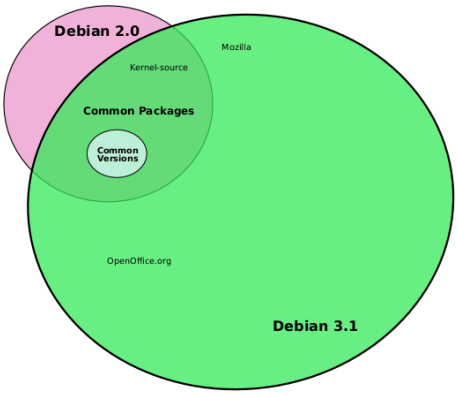


Figure 3: Illustration of common packages between Debian 2.0 and 3.1. Among these packages, we may find a subset that has the same version number.

Tables 2 and 3 contain some statistics about common packages in different stable versions. As explained above, we assume that two versions have a package in common if that package is included in both, independently of the version number of the package. Each table displays in its second column the number of packages that a version of Debian has in common with the other versions (see column "Com pkgs"). To facilitate the comparison in relative and absolute terms, the same version of Debian that is compared is included. Needless to say, Debian 2.0 will have in common with itself 1,096 (all) source packages.

Out of the 1096 packages included in Debian 2.0 only about 800 appear in the latest version of Debian (at time of

[5]One of the anonymous workshop reviewers pointed out that this kind of distribution is common in human-created artifacts, and is often considered to be an indication of human cognitive limitations.

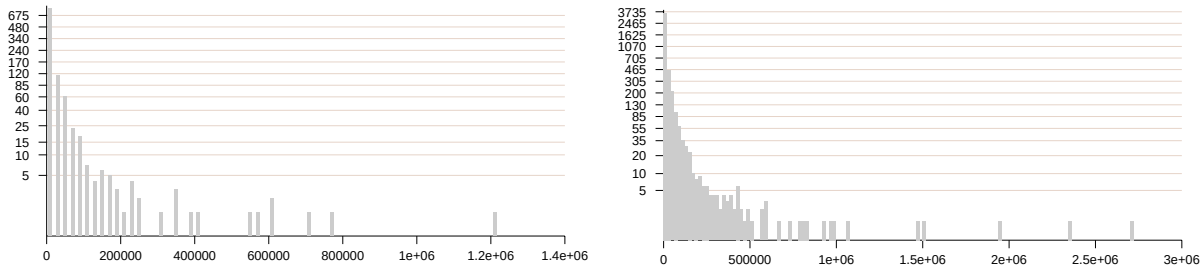


Figure 2: Histogram with the SLOC distribution for Debian packages. Left: Debian 2.0. Right: Debian 3.0

publication of this paper). This means that around 25% of the packages have disappeared from Debian in seven years. The number of packages of the 3.0 version that are still included in 3.1 is 3,848 out of 4,578 which gives us a similar percentage of *lost* packages.

If we consider those packages with version numbers that have not varied, we have to identify packages included in two different Debian versions that have the same package version (see column "Com vers."). Again, we add the own Debian version being compared. Because of that, Debian 2.0 will have all of its packages (1,096) in common with itself.

The fact that Debian 3.1 includes 158 packages that have not evolved since Debian 2.0 is very surprising, as 15% of the source packages included in Debian 2.0 have stayed almost with no alterations since they were introduced seven years ago (or earlier). As expected, the number of packages with versions in common increases for neighboring distributions.

4.4 Observations on the programming languages

Our methodology implies to identify the programming language of source code files before counting the number of SLOCs. Thanks to this, we are able to compute the significance of the different programming languages in Debian. The most used language in all Debian versions is C with percentages that vary between 55% and 85% and with a big advantage on its immediate pursuer, C++. It can be observed, nevertheless, that the importance of C is diminishing gradually, whereas other programming languages are growing at a steady rate.

For example, in table 4 the evolution of the most significant languages – those that surpass 1% of code in Debian 3.1 – is shown. Below the 1% mark we can find, in this order: tcl, Ada, PHP, Pascal, ML, Objective C, YACC, C#, Lex, Awk, Sed and Modula3.

There exist some programming languages that we could consider as minor languages and that reach a high position in the classification. This is because although being present in a reduced number of packages, these are large in size. That is the case of Ada, that sums up 430 KSLOC in three packages (gnat, an Ada compiler, libgtkada, a binding to the GTK library, and Asis, a system to manage sources in Ada) of a total of 576 KSLOC that have been identified as code written in Ada in Debian 3.0. A similar case is the one for Lisp, that counts with more than 1.2 MSLOC only for GNU Emacs and XEmacs of around 4 MSLOC in the whole distribution.

The programming language distribution pie-charts display a clear tendency in the decline in relative terms of C. Something similar seems to happen to Lisp, which was the third most used language in Debian 2.0 and has become the fifth in Debian 3.1 (in fact, in 3.1, the forth language is Perl), and that foreseeably will continue backing down in the future. In contrast, the part of the pie corresponding to C++, shell and other programming languages grows.

Figure 5 provides the relative evolution of programming languages which gives a new perspective of the growth for the last five stable Debian versions. We therefore take the Debian 2.0 version as reference and suppose that the presence of each language in it is 100% (normalized to 1) so that growth for a programming language is shown relative to itself. The graph should be read as follows: for each line in Debian 2.0 for a given language, the figure gives the number of lines in subsequent Debian releases for that language.

Previous pies evidenced that C is backing down as far as its relative importance is concerned. In this one we can observe that in absolute terms C has grown more than 300% throughout the four versions (see figure 4 for a histogram with absolute values). But we can see that scripting languages (shell, Python and Perl) have undergone an extraordinary growth, all of them multiplying their presence by factors superior to seven, accompanied by C++. Languages that grow a smaller quantity are the *traditional*, compiled ones (Fortran and Ada) and others (such as Lisp, a *traditional* language that does not require compilation). This can give an idea of the importance that interpreted languages have begun to have in the libre software world.

Figure 5 includes the most representative languages in Debian, but excludes Java and PHP, since the growth of these two has been enormous, in part because their presence in Debian 2.0 was testimonial, in part because their popularity in the latest time is beyond doubt.

4.5 Observations on the file sizes

It should be remarked that some of the most important programming languages have spectacular increases in their use, but that their mean file sizes remain generally constant (see table 5). Thus, for C the average length lies around 260 to 280 SLOC per file, whereas in C++ this value is located in an interval going from 140 to 185. We can find the exception to this rule in the shell language, that triples its mean size. This may be because the shell language is very singular: almost all the packages include something in shell for their installation, configuration or as *glue*. It is probable that this type of scripts get more complex and thus grow over the years.

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	1,096	1,096	25,267,766	110,587	25,267,766
Debian 2.1	1,066	666	11,518,285	11,5126	26,515,690
Debian 2.2	973	367	3,538,329	86,810	19,388,048
Debian 3.0	754	221	1,863,799	70,326	15,888,347
Debian 3.1	813	158	1,271,377	15,296	15,594,976

Table 2: Packages and versions in common for Debian 2.0

Version	Com pkgs	Com vers.	SLOC com vers.	Files com vers.	SLOC com pkgs
Debian 2.0	813	158	1,271,377	15,296	15,594,976
Debian 2.1	1,124	231	2,306,969	27,543	23,630,211
Debian 2.2	1,946	508	4,992,308	60,525	36,584,110
Debian 3.0	3,848	1,567	16,042,810	211,299	78,451,818
Debian 3.1	8,560	8,560	215,812,764	931,834	215,812,764

Table 3: Packages and versions in common for Debian 3.1

	2.0	% 2.0	2.1	% 2.1	2.2	% 2.2	3.0	% 3.0	3.1	% 3.1
C	19,371	76.7%	27,773	74.9%	40,878	69.1%	66.6	63.1%	120.5	55.8%
C++	1,557	6.2%	2,809	7.6%	5,978	10.1%	13.1	12.4%	36.4	15.8%
Shell	645	2.6%	1,151	3.1%	2,712	4.6%	8.6	8.2%	20.4	9.4%
Perl	425	1.7%	774	2.1%	1,395	2.4%	3.2	3.0%	6.4	2.9%
Lisp	1,425	5.6%	1,892	5.1%	3,197	5.4%	4.1	3.9%	6.8	3.1%
Python	122	0.5%	211	0.6%	349	0.6%	1.5	1.4%	4.1	1.9%
Java	22	0.1%	58	0.2%	183	0.3%	0.5	0.5%	3.8	1.7%
Fortran	494	2.0%	735	2.0%	1,182	2.0%	1,939	1.8%	2.7	1.3%

Table 4: Top programming languages in Debian. For Debian 2.0, 2.1 and 2.2 the sizes are given in KSLOC, for versions 3.0 and 3.1 in MSLOC.

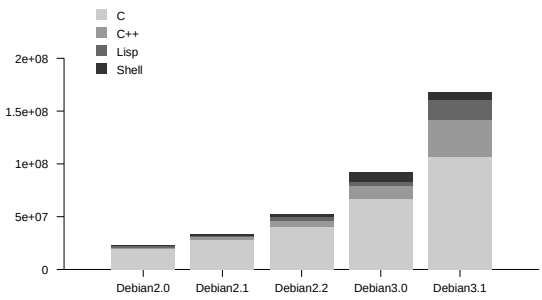


Figure 4: Evolution of the four most used languages in Debian.

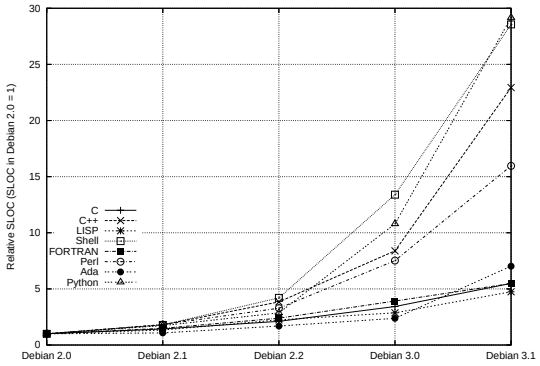


Figure 5: Relative growth of some programming languages in Debian.

It is very peculiar to see how structured languages usually have larger average file lengths than object-oriented languages. Thus the files in C (or Yacc) usually have higher sizes, in average, than those in C++. This makes us think that modularity of programming languages is reflected in the mean file size.

5. CONCLUSIONS AND FURTHER RESEARCH

In this paper we have shown the results of a study on the evolution of the stable versions of Debian from the year 1998 onwards. We have traced and presented the evolution of the size of its source code (measured in physical source lines of code), of the number and size of the packages, and of the use of the various programming languages.

Among the most important evidence we have found we can highlight the drastic evolution rate of the distribution: stable versions double in size (measured by number of packages or by lines of code) approximately every two years. This, when combined with the huge size of the system (about 200 MSLOC and 8,000 packages in 2005) may pose significant

Lang.	Deb. 2.0	Deb. 2.1	Deb. 2.2	Deb. 3.0	Deb. 3.1
C	262.88	268.42	268.64	283.33	276.36
C++	142.50	158.62	169.22	184.22	186.65
Lisp	394.82	393.99	394.19	383.60	349.56
shell	98.65	116.06	163.66	288.75	338.25
Yacc	789.43	743.79	762.24	619.30	599.23
Mean	228.49	229.92	229.46	243.35	231.6

Table 5: Mean file size for some programming languages.

problems for the management of the future evolution of the system, something that has probably influenced the delays in the release process of the last stable versions.

A specific problem in this realm comes from the fact that until now the mean size of packages has remained almost constant, which means that the system has more and more packages (growing linearly with the size of the system in SLOCs). Since there is a certain level of complexity related to the specifics of each package, which imposes a limit on the number of packages per developer, this means that the project would need to grow in terms of developers at the same pace. However, such a growth is not easy, and causes problems of its own, specially in the area of coordination.

With respect to the absolute figures, it can be noted that Debian 3.1 is probably one of the largest coordinated software collections in history, and almost for sure the largest one in the domain of general-purpose software for desktops and servers. This means that the human team maintaining it, which has also the peculiarity of being completely formed by volunteers, is exploring the limits of how to assemble and coordinate such a huge quantity of software. Therefore, the techniques and processes they employ to maintain a certain level of quality, a reasonable speed of updating, and a release process that delivers stable versions quite usable, are worth studying, and can for sure be of use in other domains which have to deal with large, complex collections of software.

As far as the programming languages are concerned, C is the most used language, although it is gradually losing importance. Scripting languages, C++ and Java are those that seem to have a higher growth in the newer releases, whereas the traditional compiled languages have even inferior growth rates than C. These variations also imply that the Debian team has to include developers with new skills in programming languages in order to maintain the evolving proportions. By looking at the trends in languages use withing the distribution, the project could estimate how many developers fluent in a given language it will need. In addition, this evolution of the different languages can also be considered as an estimation of how libre software is evolving in terms of languages used, although some of them are for sure misrepresented (for instance, Java is underrepresented, possibly because of licensing issues).

The evolution shown in this paper should also be put in the context of the activity of the volunteers doing all the packaging work. While some work has been done in this area [9], more research needs to be performed before a link can be established between the evolution of the skills and size of the developer population, the complexity and size of the distribution, the processes and activities performed by the project, and the quality of the resulting product. Only by understanding the relationships between all these parameters can reasonable measures be proposed to improve the quality of the software distribution, or shorten the release cycle without harming reliability and stability of the releases.

All in all, the study of distributions such as Debian can be of great interest not only for understanding their evolution, but also to be used as good case studies which can help to understand large, complex software systems which are more and more common in many domains.

6. REFERENCES

[1] J. J. Amor, J. M. Gonzalez-Barahona, G. Robles, and I. Herraiz. Measuring libre software using Debian 3.1 (Sarge) as a case study: preliminary results. Upgrade Magazine, Aug. 2005.

[2] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth. Software evolution observations based on product release history. In *Proc Intl Conference on Software Maintenance*, pages 160-170, 1997.

[3] M. W. Godfrey and Q. Tu. Evolution in Open Source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131-142, San Jose, California, 2000.

[4] J. M. Gonzalez-Barahona, M. A. Ortuno Perez, P. de las Heras, J. Centeno, and V. Matellan. Counting potatoes: the size of Debian 2.2. *Upgrade Magazine*, II(6):60-66, Dec. 2001.

[5] M. M. Lehman and L. A. Belady, editors. Program evolution: Processes of software change. *Academic Press Professional*, Inc., San Diego, CA, USA, 1985.

[6] M. M. Lehman and J. F. Ramil. Implications of laws of software evolution on continuing successful use of cots software. *Technical report*, Imperial College, 1998.

[7] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15-44, 2001.

[8] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS'97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, nov 1997.

[9] M. Michlmayr and B. M. Hill. Quality and the reliance on individuals in free software projects. In *Proceedings 3rd Workshop on Open Source Software Engineering*, pages 105-109, Portland, USA, 2003.

[10] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Proceedings of the International Workshop on Principles in Software Evolution*, pages 165-174, Lisbon, Portugal, September 2005.

[11] G. Succi, J. W. Paulson, and A. Eberlein. Preliminary results from an empirical study on the growth of open source and commercial software products. In *EDSER-3 Workshop*, Toronto, Canada, May 2001.

[12] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International conference on Software Engineering*, pages 492-497, 1976.

[13] W. M. Turski. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering*, 22(8):599-600, 1996.

[14] D. A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size. *Technical report*, June 2001.

Scenarios for Mining the Software Architecture Evolution

Yaojin Yang
Nokia Research Center
P.O. Box 407, FIN-00045
+358718008000
yaojin.yang@nokia.com

Claudio Riva
Nokia Research Center
P.O. Box 407, FIN-00045
+358718008000
claudio.riva@nokia.com

ABSTRACT

In this position paper, we introduce our latest activities on architecture evolution analysis through software repository mining. The traditional approaches for software repository mining provide means for analyzing source-level information. However, we believe that software repository mining can also provide valuable results for analyzing the system evolution at the architectural level.

There are two challenges for analyzing the architecture evolution. The first one is to have in place a process for recovering the architectural models of the various releases. Architecture evolution is often visible only in the evolution of the implementation and this complicates the monitoring process. The second one is to have access to the past design models that were created by the architects during the design phase. A practical solutions for versioning the architectural models is not in use yet and this complicates the possibility of accessing the past design decisions.

Analyzing architecture evolution through software repository mining represents the most promising choice. In order to conduct the analysis through software repository mining, we introduce our meta-model covering the design and implementation spaces. Then, we define a set of scenarios that demonstrate the architecturally significant analysis that we can conduct by mining the software repository.

Categories and Subject Descriptors

D.2.11 Software Architectures, D.2.13 Reusable Software

General Terms: Documentation, Experimentation

Keywords: Architecture evolution, Mining software repository, Architecture recovery

1. INTRODUCTION

The software architecture evolution typically happens on two parallel tracks: the design space and the implementation space. While the evolution on the design space concerns the intentions of the designers, the evolution on the implementation space can have deep implications at the architectural level. From our experience, understanding the evolution at the implementation level can help to understand the evolution at the architecture level where it is harder to monitor and trace the changes. For this reason, it is not easy to keep the architecture models up to date. Therefore, providing support to deal with this issue is very important from the perspective of architecture evolution.

We provide an example of architecture evolution triggered by the implementation that is frequently happening in the lifecycle of software platforms like the ones developed in Nokia. We monitor and analyze such evolution through mining software repository.

We consider a *binary component* (like a DLL) that belongs to the implementation space. If the source files associated with the binary component have been modified, we say that the binary component itself is modified and it is evolving. The evolution of the binary component may have implications in the architecture space, i.e. in the architecture design of the system. If the modifications in the binary component affect the way the component interacts with the environment (e.g. using a new interface), we can say that also the *logical component* in the design space has changed and the architecture of the system has also evolved.

We highlight that there is not a direct link between the changes in the implementation with the changes in the architecture. Only some implementation-level changes have an effect on the architecture and we call them *architecturally significant*. The main focus of our work is to study the architecturally significant changes for a software system.

This is our approach for monitoring the architecture evolution by mining software repositories. First, for each release we build an implementation and design combined architecture model according to a defined meta-model by using our reverse architecting environment ([2] and [6]) and import the models into our software repository. Second, we compare the models' implementation spaces between release *2* and its previous release *1* through mining the software repository. Third, if binary component evolution is identified, we trace up to the design spaces of both releases *1* and *2* and identify the parent logical component of evolved binary component. Fourth, we compare the topology of the graphs based on the parent logical component between the models' design spaces of release 1 and release 2. If there are not identical, we claim that there is implementation trigged evolution happening on the parent logical component.

In [3], Koschke and Simon propose an approach to map the design and the implementation based on the same module viewtype. The difference with our work of building an implementation and design combined architecture model is that we do not assume that the viewtypes of design space and implementation space are the same. In fact, our design space is presented in component and connector viewtype and our

MSR'06, May 22–23, 2006, Shanghai, China.

implementation is presented in module viewtype. That is, our model is presented in a combined viewtype.

For supporting architecture evolution analysis through software repository mining, we utilize our existing reverse architecting environment [2] & [6]. The *reverse architecting tool set* offered by the environment provides us a tool chain from source code analysis till model abstraction for recovering architecture models. Columbus [5] tool is deployed in the tool chain as source code analyzer. *MySQL database* is integrated into the environment, which is used as our software repository for storing architecture models of different releases. The environment's *model validation tool* facilitates our comparison between architecture models of different releases. The architecture evolution is identified through the comparison.

2. META-MODEL OF ARCHITECTURE MODELS

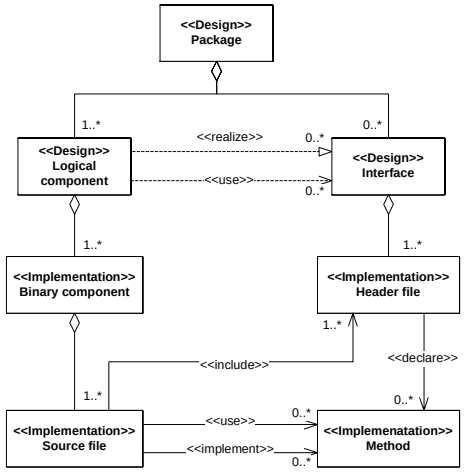


Figure 1. Meta-model for the architecture models

We have developed a simplified meta-model of the architecture that serves the purposes of studying the architectural evolution. The meta-model captures the design and implementation aspects, as shown in Figure 1. The meta-model provides a traceability mechanism between the architecture design and the implementation. This represents a key element for studying the software architecture evolution.

We make a distinction between the design and implementation space. In the design space, a *package* contains one or more *logical components* and a set of *interfaces*. The *logical components* implement the functionality of the system. They can realize or/and depend on any number of *interfaces*.

In the implementation space, the *binary component* is the aggregation of several *source files*. The *source files* include one or more *header files*. The *header files* declare the methods that are implemented in or used by a *source file*.

3. APPROACH FOR MODEL RECOVERY

In our architecture models, the instances of elements and relations presented in the meta-model are mostly captured or abstracted from implementation. However, instances of *aggregation* between *logical component* and *binary component* and *aggregation* between *interface* and *header file* are directly extracted from design.

We use Columbus for capturing model elements and relations in the implementation space. The initial resulting model conforms to the FAMIX meta-model [1] (Table 1). Then, we filter information that is not defined by the meta-model (Figure 1). Since, *binary component* and *aggregation* between *binary component* and *source file* are specified in specific project file, the Columbus has been customized in order to extract such information and import it into the implementation model.

In the design space, model elements of documented *logical component*, *interface* and *package*, *aggregation* between *package* and *logical component*, and *aggregation* between *package* and *interface* are captured by parsing architectural logical view and interface specification document.

In order to merge the implementation model and the design model to form a complete architecture model defined by the meta-model (Figure 1), *aggregation* between *logical component* and *binary component* and *aggregation* between *interface* and *header file* are the key relations to rely on. The *aggregation* between *logical component* and *binary component* can be obtained through parsing design tables and the *aggregation* between *interface* and *header file* can be obtained through parsing interface specification document. If certain *binary component* or *header file* doesn't belong to any *aggregation*, it is usually the case that new *logical component* or *interface* is added into the design but is not documented.

The *dependency* and *realization* between *logical component* and *interface* are considered as key measurements for monitoring implementation trigged architecture evolution. Therefore, they have to be abstracted from implementation.

Entities	
Entity	**Description**
`Class`	The definition of the class
`Method`	The definition of method of a class
`Attribute`	The definition of an attribute of a class
`Function`	The definition of a function or a procedure that has a global visibility
`Macro`	A C++ macro definition with `#define`
`TypeDef`	A C++ type definition with `typedef`
`GlobalVariable`	The definition of a global variable
`File`	A source file
`Directory`	A directory in the file system
`Package`	A Java package
Relations	
Relation	**Description**
has_method	A class declares a method
has_attribute	A class declares an attribute
has_class	A class declares a nested class
inherit	A class inherits from another class
invocation	A method or a function invokes a method or a function
access	A method or a function access an attribute or a global variable
include	A source or header file includes an header file
expansion	A function or a method expands a macro
use_type	A function or a method use a user's defined type
contain	A file contains the definition of a class, a macro, a type definition and a global variable
implement	A file defines the implementation of a method
decl_fn	A file declares a function
def_fn	A file defines a function
contain_file	A directory contains a file
contain_dir	A directory contains another directory
pkg_contain	A Java package contains a class

Table 1. The FAMIX meta-model

4. CHARACTERIZING THE EVOLUTION OF THE SOFTWARE MODELS

During the evolution of the software system, both the design and the implementation spaces are modified. The design space is modified by the software designers according to the requirements of the system. The implementation space is modified by the programmers who are implementing new features or modifying the existing code.

The implementation is driven by the design but not all the changes in the implementation are reflected by the design (as we have discussed in [2]). Moreover, the versioning of the design models is not yet well understood and the practice shows that tracing the modifications from one design to the next one is not an easy task. As a result, the only reliable information about the evolution of the system is mainly visible in the implementation space.

We need to link the evolution in the design space with the evolution in the implementation.

We characterize the evolution of the system by only comparing the topology of the software models.

4.1 The evolution of the implementation space

Changes in the implementation space are identified by changes in the topology of the graphs that we extract with the source code analyzers.

A *build component* is changed when one of the following items has been modified between two different releases:

- The set of *source files* that belong to the *build component*
- The set of *use* relations between a *source file* and a *method*
- The set of *implement* relations between a *source file* and a *method*
- The set of *include* relations between the *source files* in the *build component* and the *header file*

A *header file* is changed when the method declarations have been modified (e.g. when method declarations have been added, removed or changed).

We note that we ignore those modifications that do not modify the topology of the graphs.

4.2 The evolution of the design space

We directly link the evolution of the elements in the design space with the modifications that happen in the implementation space. We define the following rules:

1. One *logical component* is changed when at least one of its *binary components* have been changed, removed or new ones have been added.
2. One *interface* is changed when at least one of its *header files* have been changed, removed or new ones have been added. It is not possible to freeze the interfaces but they can evolve in the same way like components.
3. One *package* is changed when (1) one containing element (either a component or an interface) has changed, (2) a new element has been added or (3) an existing element has been removed.

5. SCENARIOS OF ARCHITECTURE EVOLUTION ANALYSIS

We present common scenarios of architecture evolution analysis. The scenarios are listed according to their impact on the overall architecture (from high to low impact). In Table 2, we analyze the relations between the types of architecture evolution and scenarios.

Adding one new feature

One typical scenario is to add a new feature in the system. This activity typically involves creating new interfaces, modifying existing interfaces and introducing new logical or/and binary components. Adding a new feature can have a large impact on the overall architecture.

The modification of the existing interfaces may impact the functionality of existing binary components. However, not all the cases can be predicted at design time and only during the implementation problems may arise. After the implantation, it is important to monitor what are the effects of these changes on the overall design.

The new logical or/and binary components may also create unexpected dependencies that are discovered only during the implementation. It is important to detect these architectural changes.

Restructuring the design

Improving the overall design is a preventive maintenance activity that can have large impacts on the system. The designers should be able to monitor how these activities can affect the various logical or/and binary components, how often they are happening and if certain logical or/and binary components are often modified.

Modifying one binary component

When one binary component is modified to extend its functionality the changes may impact other binary components or/and even logical components. It is important to control the effects of the changes.

Studying the evolution of one logical component

Studies on the evolution of particular logical components are typically conducted to assess their quality, stability and to identify the weaknesses. By studying the evolution at the architectural level we may be able to reveal unfavorable patterns of evolutions like too frequent changes or changes with too big effects on the rest of the system. The studies may lead to restructure the logical component.

Monitoring the evolution of the interfaces

Interfaces cannot be frozen but they are evolving. In most cases, new interfaces are added to provide access to new functions. Ideally interface should be kept stable, but modifying interfaces is sometimes required by the modification of implementation. However, removing interface is not a common case.

Fixing a bug in the system

We expect that bug fixing is not causing big impacts at the architectural level. Bug fixing should only be limited by modifying the internal implementation of the binary components but not their external dependencies. If there are architectural modifications happening because of bug fixing, then there may be fundamental problems in the implementation and the current design should be revisited.

Correlating software metrics with the architectural evolution

Several software metrics are calculated by the Columbus tool. We need to correlate the trends of the software metrics with the changes in the architecture. This will enable us to monitor the effect of certain architectural changes on the quality of the software.

	Design trigged evolution	Implementation trigged evolution
Adding one new feature	**Happen**	**Maybe happen**
Restructuring the design	**Happen**	**Maybe happen**
Modifying one binary component	**Not happen**	**Happen**
Studying the evolution of one logical component	**Happen**	**Maybe happen**
Monitoring the evolution of the interfaces	**Happen**	**Not happen**
Fixing a bug in the system	**Not happen**	**Maybe happen**
Correlating software metrics with the architectural evolution	**Happen**	**Happen**

Table 2. Types of architecture evolution in the scenarios

6. CONCLUSIONS

In this paper, we presented our work of analyzing software architecture evolution through mining software repository. Our position is that the software repositories contain valuable information for monitoring the architecture evolution but this information is not ready available. In order to make it more explicit, we need to isolate the architecturally significant changes that happen in the implementation and have an impact on the design space.

In the future, we will focus on providing concrete examples of analysis of the evolution of very large software systems (containing tens of millions of lines of code). One of our goal is to define architecture evolution patterns, especially implementation trigged evolution, to provide tool support for monitoring and analyzing the evolution, and ultimately to refine our architecture design so that the implementation evolution will have minimum impact on the architecture level.

7. REFERENCES

[1] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 – the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[2] C. Riva, P. Selonen, T. Systäa, and J. Xu. UML-based reverse engineering and model analysis approaches for software architecture maintenance. *In Proc. of the The 20th IEEE International Conference on Software Maintenance*, Chicago, Illinois, USA, September 11th - 17th 2004. IEEE Computer Society, 2004.

[3] R. Koschke R. and D. Simon, Hierarchical Reflexion Models, *In Proc. of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, 13-16 November 2003, Victoria, Canada, IEEE Computer Society Press, 2003, 36-45.

[4] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. *In Proc. of 4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004)*, 12-15 June 2004, Oslo, Norway, pages 122–132. IEEE Computer Society, 2004.

[5] Ferenc, R.; Beszédes A.: Gyimóthy T., Extracting facts with Columbus from C++ code, *In Proc. of Proceedings. 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, Tampere, Finland, March 24-26, 2004.

[6] Riva, C.; Selonen, P.; Systa, T.; Tuovinen, A.-P.; Xu, J.; Yang, Y., Establishing a software architecting environment. *In Proc. of the 4th Working IEEE / IFIP Conference on Software Architecture*, Oslo, Norway, July 12-15 2004.

Productivity Analysis of Japanese Enterprise Software Development Projects

Masateru Tsunoda
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
masate-t@is.naist.jp

Akito Monden
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
akito-m@is.naist.jp

Hiroshi Yadohisa
Doshisha University
1-3 Miyakodani, Tatara, Kyotanabe,
Kyoto, 610-0394 Japan
hyadohis@mail.doshisha.ac.jp

Nahomi Kikuchi
Software Engineering Center
Information-technology Promotion Agency
2-28-8, Hon-Komagome,
Bunkyo-ku, Tokyo, 113-6591 Japan
n-kiku@ipa.go.jp

Ken-ichi Matsumoto
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
matumoto@is.naist.jp

ABSTRACT
To clarify the relation between controllable attributes of a software development and its productivity, this paper experimentally analyzed a software project repository (SEC repository), consisting of 253 enterprise software development projects in Japanese companies, established by Software Engineering Center (SEC), Information-technology Promotion Agency, Japan. In the experiment, as controllable attributes, we focused on the outsourcing ratio of a software project, defined as an effort outsourced to subcontract companies divided by a whole development effort, and on the effort allocation balance among development phases. Our major findings include both larger outsourcing ratio and smaller upstream process effort leads to worse productivity.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *Cost estimation, productivity*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management – *Strategic information systems planning;*

General Terms: Management, measurement, economics

Keywords
Software productivity, subcontract, upstream process, custom software, software project repository

1. INTRODUCTION
Estimation of software development effort is required throughout a software development lifecycle to set up, evaluate and revise a project plan including resource allocation and scheduling. Software productivity is one of the key factors in drawing up the early effort estimation in a project, although the productivity greatly varies according to project attributes such as business area, application type, programming language, team size and experience, etc [8]. Past researches have shown that some attributes do affect the productivity [2][7][8][9]. Unfortunately, many of these attributes, e.g. business area and programming language, are usually not controllable by a software development company.

This paper focuses on the project attributes that are controllable by a software development company. Through the analysis of the year 2005 version of the SEC repository consisting of 253 samples of enterprise software development in Japanese companies, this paper seeks to clarify the relation between project attributes and software development productivity. The repository has been developed and maintained by Software Engineering Center (SEC), Information-technology Promotion Agency, Japan [10].

In the analysis, as controllable project attributes, this paper focuses on the outsourcing ratio of a software project, and on the effort allocation balance among development phases. While high outsourcing ratio is one of the major distinctive properties of Japanese enterprise software development, there is no past researches focused on the relation between outsourcing ratio and productivity of software development in Japan.

The reminder of this paper first describes details of the SEC repository we used (Section 2). Next, describes analyses we conducted to clarify the relation between project attributes and productivity (Section 3). Afterward, related works are described (Section 4); and in the end, conclusions and future topics will be shown (Section 5).

2. THE SEC REPOSITORY
The year 2005 version of the SEC repository consists of 1009 software projects held in 15 Japanese companies. These projects are custom enterprise software, which is the majority in Japan [4][5]. Each project is characterized by about 400 attributes (metrics); however, 87.7% of them were unrecorded on average. Since the project size is a necessary attribute to calculate the productivity, we excluded projects having missing value in the function

MSR'06, May 22–23, 2006, Shanghai, China.

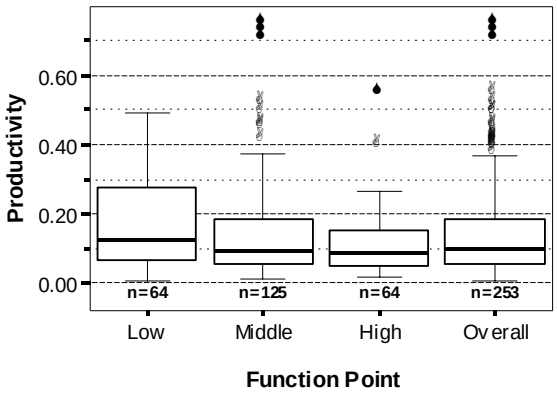


Figure 1. Boxplots of productivity for project groups classified by FP

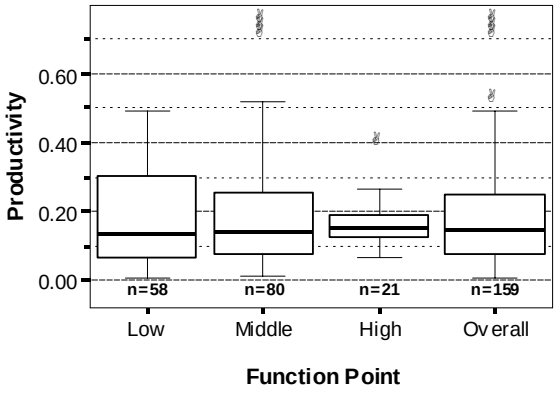


Figure 2. Boxplots of productivity for project groups classified by FP (Outsourcing ratio is zero)

Table 1. P-values for project groups classified by FP

Group 1	Group 2	All	Outsourcing ratio is zero
Low	Middle	0.072	0.789
Low	High	*0.017*	0.610
Middle	High	0.296	0.744

point (FP) attribute from our analysis. Also, we excluded maintenance projects and enhancement projects, and focused to new development projects and re-development projects because maintenance/enhancement processes are often very different from new/re-development processes. As a result, 253 projects were selected for the analysis. All these projects were waterfall process development.

Productivity is defined as FP divided by (total) development effort (person-hour). FP methods consist of IFPUG (34%), SPR (26%), NESMA (3%), others (31%), and missing (5%). Productivity of the upper quartile project is about 3.3 times larger than the lower quartile project. Because of biased distribution of productivity and other metrics, we used the median of metrics instead of the average in our analysis. We also used non-parametric tests in the analysis.

In the analysis, we mainly focused to two factors to analyze productivity. One is the outsourcing ratio, defined as an effort outsourced to subcontract companies divided by a whole development effort. The other is the upstream process ratio, defined as the sum of requirement analysis process effort and design process effort divided by a whole development effort. In many cases, these two factors are controllable by a software development company.

3. PRODUCTIVITY ANALYSIS

There are several factors that may influence software productivity. One of the most considerable factors is the project size [1][3] since development processes (e.g. human resource allocation) vary according to the project size. Therefore, we also focus to the FP, which is one of project size metrics, in addition to the outsourcing ratio and the upstream process ratio.

3.1 Project Size

Before focusing on the outsourcing ratio and the upstream process ratio, we analyzed the influence of FP (project size), which is the most basic factor in productivity analyses.

In this analysis projects were classified into three groups by their FP. Projects whose FPs are equal to or less than lower quartile were classified as the “low” group. Projects whose FPs are equal to or greater than upper quartile were classified into the “high” group. The rest projects were classified into the “middle” group.

To visually explore the difference in productivity among three groups, boxplots were used. Figure 1 shows boxplots of productivity for the three project groups (FP=low, FP=middle, and FP=middle) plus “overall” group for whole projects. Circles indicate outliers, and stars indicate extreme outliers. Upper quartile of FP is about 4.1 times larger than lower quartile. The figure shows there are several high productivity projects in the low group. Median of productivity of the low group is about 1.4 times larger than the high group. Median of productivity of the middle group is only about 1.1 times larger than the high group. Using Mann-Whitney U test, we confirmed that the difference of productivity is statistically significant at the 0.05 level between the low group and the high group. P-values are shown in the column “All” of Table 1. The column “Group 1” and the column “Group 2” indicate paired group when testing. Italic indicates p-value < 0.05.

Next we excluded the influence of outsourcing ratio since there was a high correlation between FP and outsourcing ratio (Spearman's rank correlation was 0.51). Figure 2 shows boxplots for projects whose outsourcing ratios were zero. Median of productivity is almost same among the three groups. The differences in productivity among the three groups are not statistically significant at all. P-values are shown in the column “Outsourcing ratio is zero” of Table 1.

Above all, the SEC repository showed that the project size alone do not directly influence the productivity. However, it can be said that large size projects have high outsourcing ratio; and thus, it indirectly leads to worse productivity.

3.2 Outsourcing Ratio

Figure 3 shows boxplots of productivity for three project groups classified by the outsourcing ratio. (Note that the number of projects of the low group is different from the high group because there are many projects whose outsourcing ratios are zero and all

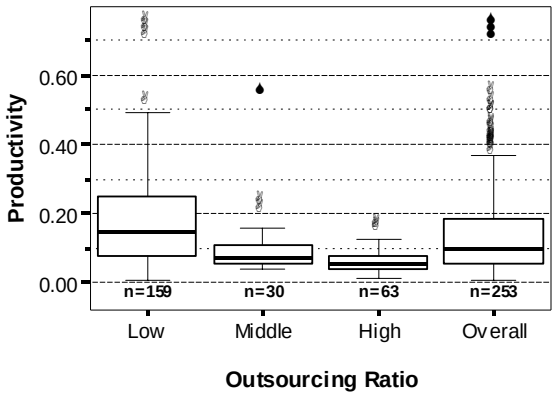


Figure 3. Boxplots of productivity for project groups classified by outsourcing ratio

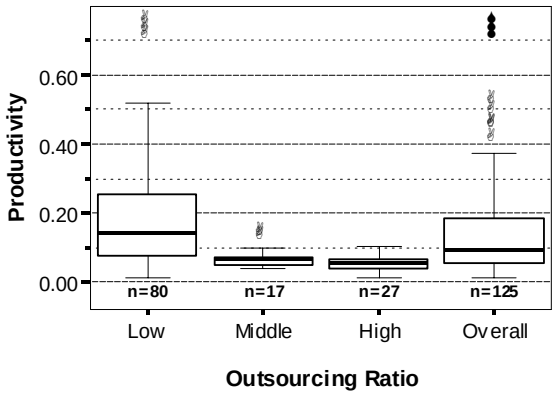


Figure 4. Boxplots of productivity for project groups classified by outsourcing ratio (FP is middle)

Table 2. P-values for project groups classified by outsourcing ratio

Group 1	Group 2	All	FP is middle
Low	Middle	*3.2E-04*	*3.9E-04*
Low	High	*8.8E-14*	*1.8E-07*
Middle	High	*0.002*	0.104

of them are classified as the lower quartile group.) The figure shows there are high productivity projects in the low group. Median of productivity of the low group is about 2.6 times larger than the high group. The differences of productivity are statistically significant at the 0.05 level among the three groups. P-values are shown in the column "All" of Table 2.

Next we excluded the influence of FP. As described in Section 3.1, high FP projects tend to have high outsourcing ratio. Figure 4 shows boxplots for projects whose FPs are in the "middle" group. Figure 4 shows similar tendency to Figure 3. The differences in productivity are statistically significant at the 0.05 level between the low group and other groups. P-values are shown in the column "FP is middle" of Table 2.

These results suggest that lower outsourcing ratio projects have higher productivity. We consider that higher outsourcing ratio introduces communication overhead between companies. However, these results do not mean that the outsourcing ratio must be suppressed because the development cost would be higher if a company stopped outsourcing. (Unfortunately, because the SEC repository does not record cost factors, we were not able to analyze relations between cost and outsourcing ratio.) In addition, if the software size is too large to be developed in-house, a company needs to outsource a part of software development. Anyway, a company must be aware of the trade-offs between increase of effort and decrease of cost.

3.3 Upstream Process Ratio

Figure 5 shows boxplots of productivity for three project groups classified by upstream process ratio. The figure shows there are high productivity projects in the high group. Median of productivity of the high group is about 1.8 times larger than the low group. The differences in productivity are statistically significant at the 0.05 level between the high group and other groups. P-vales are shown in the column "All" of Table 3.

Next we excluded the influences of FP and outsourcing ratio. Figure 6 shows boxplots for projects whose FPs are in the "middle" group. The differences of productivity are statistically significant at the 0.05 level between the high group and other groups. P-values are shown in the column "Function point is middle" of Table 3. Similarly, Figure 7 is boxplots of projects whose outsourcing ratio is zero. The differences in productivity are not statistically significant at the 0.05 level among the three groups. P-values are shown in the column "Outsourcing ratio is zero" of Table 3. Although there was no strong significance among these groups, significance p=0.083 (< 0.10) was observed between middle and high group.

From these results, high upstream process ratio alone has some influence to the productivity. This suggests that high upstream process ratio may avoid additional effort (reworks) on downstream processes.

4. RELATED WORK

Maxwell et al. [8] and Premraj et al. [9] analyzed the influence of the business sector type on productivity, using Finnish software development project dataset collected by Software Technology Transfer Finland (STTF). Lokan et al. [6] also showed productivity analysis focused on the business sector using a dataset of International Software Benchmarking Standards Group (ISBSG). In these researches, projects in the manufacturing sector have the highest productivity, and projects in the banking/Insurance sector have the lowest productivity. Same tendency was observed in the SEC repository.

Blackburn et al. [2] analyzed the influence of requirement analysis process ratio on the productivity. They found that requirement analysis process ratio leads to higher productivity of coding process. Our finding about the influence of upstream ratio follows their result.

5. CONCLUSIONS

We analyzed the influences of outsourcing ratio and upstream process ratio on the development productivity of Japanese enterprise software development projects. We found that both lower outsourcing ratio projects and higher upstream process ratio pro-

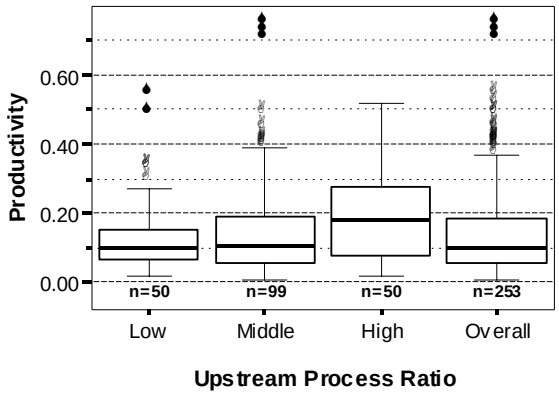


Figure 5. Boxplots of productivity for project groups classified by upstream process ratio

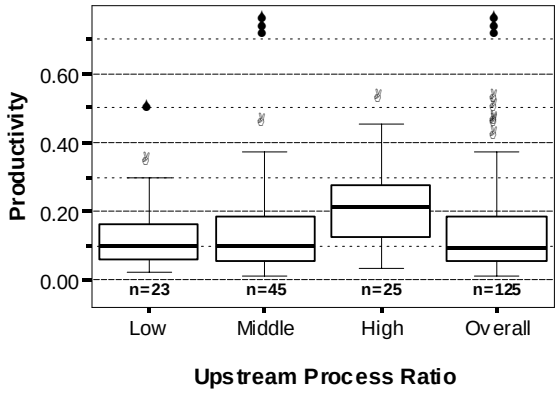


Figure 6. Boxplots of productivity for project groups classified by upstream process ratio (FP is middle)

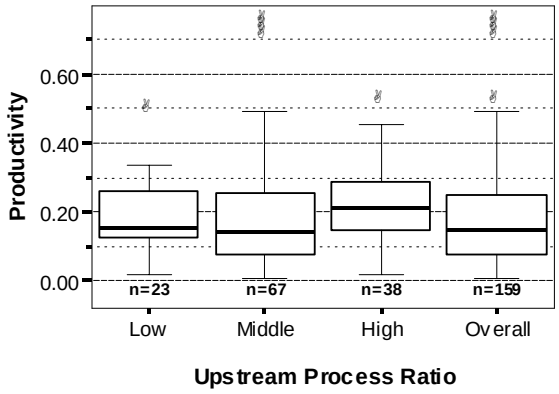


Figure 7. Boxplots of productivity for project groups classified by upstream process ratio (Outsourcing ratio is zero)

Table 3. P-values for project groups classified by upstream process ratio

Group 1	Group 2	All	FP is middle	Outsourcing ratio is zero
Low	Middle	0.819	0.974	0.595
Low	High	*0.009*	*0.016*	0.222
Middle	High	*0.012*	*0.011*	0.083

jects have significantly higher productivity. Our future work is to analyze the influences of other factors thoroughly.

6. ACKNOWLEDGMENTS

The authors would like to thank Software Engineering Center, Information-technology Promotion Agency, Japan for offering the SEC Repository. This work is supported by the EASE (Empirical Approach to Software Engineering) project of the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

7. REFERENCES

[1] Banker, R., Chang, H., and Kemerer, C. Evidence on economies of scale in software development. *Information and Software Technology, 36*, 5 (1994), 275-282.

[2] Blackburn, J., Scudder, G., and Wassenhove. L. Improving Speed and Productivity of Software Development: A Global Survey of Software Developers. IEEE Trans. on Software Eng., 22, 12 (1996), 875-885.

[3] Boehm, B. *Software Engineering Economics*. Prentice Hall, 1981.

[4] Cusumano, M., MacCormack, A., Kemerer, C., and Crandall, B. Software Development Worldwide: The State of the Practice. *IEEE Software, 20*, 6 (2003), 28-34.

[5] Duvall, L. A study of software management: The state of practice in the United States and Japan. *Journal of Systems and Software, 31*, 2 (1995), 109-124.

[6] Lokan, C., Wright, T., Hill, P., and Stringer, M. Organizational Benchmarking Using the ISBSG Data Repository. *IEEE Software, 18*, 5 (2001), 26-32.

[7] Maxwell, K., Wassenhove, L., and Dutta, S. Software Development Productivity of European Space, Military, and Industrial Applications. *IEEE Trans. on Software Eng., 22*, 10 (1996), 706-718.

[8] Maxwell, K., and Forselius, P. Benchmarking Software-Development Productivity. *IEEE Software, 17*, 1 (2000), 80-88.

[9] Premraj, R., Shepperd, M., Kitchenham, B., and Forselius, P. An Empirical Analysis of Software Productivity over Time. In *Proceedings of 11th IEEE International Software Metrics Symposium (METRICS'05)* (Como, Italy, September), 2005, 37.

[10] Software Engineering Center, Information-technology Promotion Agency, Japan *The 2005 White Paper on Software Development Projects (in Japanese)*. Nikkei Business Publications, 2005.

Coupling and Cohesion Measures for Evaluation of Component Reusability

G. Gui
Department of Computer Science
University of Essex, Colchester,
CO4 3SQ, UK
Tel: +44 1206 873805
ggui@essex.ac.uk

P. D Scott
Department of Computer Science
University of Essex, Colchester,
CO4 3SQ, UK
Tel: +44 1206 872015
scotp@essex.ac.uk

ABSTRACT
This paper provides an account of new measures of coupling and cohesion developed to assess the reusability of Java components retrieved from the internet by a search engine. These measures differ from the majority of established metrics in two respects: they reflect the degree to which entities are coupled or resemble each other, and they take account of indirect couplings or similarities. An empirical comparison of the new measures with eight established metrics shows the new measures are consistently superior at ranking components according to their reusability.

Categories and Subject Descriptors
D.2.8.3 [**Metrics**]: Complexity measures.

General Terms
Measurement, Experimentation.

Keywords
Coupling, Cohesion, Reusability

1. INTRODUCTION
The work reported in this paper arose as part of a project that retrieves Java components from the internet [1]. However, components retrieved from the internet are notoriously variable in quality. It seems highly desirable that the search engine should also provide an indication of both how reliable the component is and how readily it may be adapted in a larger software system.

A well designed component, in which the functionality has been appropriately distributed to its various subcomponents, is more likely to be fault free and easier to adapt. Appropriate distribution of function underlies two key concepts: coupling and cohesion. Coupling is the extent to which the various subcomponents interact. If they are highly interdependent then changes to one are likely to have significant effects on others. Hence loose coupling is desirable. Cohesion is the extent to which the functions performed by a subsystem are related. If a subcomponent is responsible for a number of unrelated functions then the functionality has been poorly distributed to subcomponents. Hence high cohesion is a characteristic of a well designed subcomponent.

We decided that the component search engine should provide the quality rankings of retrieved components based on measures of their coupling and cohesion. There is a substantial literature on coupling and cohesion metrics which is surveyed in the next section. We then describe in detail the metrics we have developed which attempt to address some of the limitations of existing metrics. In particular, we consider both the strength and transitivity of dependencies. The following section describes an empirical comparison of our proposed metrics and several popular alternatives as predictors of reusability. Section 5 presents an analysis of the results which demonstrate that our proposed metrics consistently outperform the others. The paper concludes with a discussion of the implications of the research.

MSR'06, May 22-23, 2006, Shanghai, China.

2. COUPLING AND COHESION METRICS
Cohesion is a measure of the extent to which the various functions performed by an entity are related to one another. Most metrics assess this by considering whether the methods of a class access similar sets of instance variables. Coupling is the degree of interaction between classes. Many researches have been done on software metrics [8], the most important ones are selected used in our comparative study. Table 1 and Table 2 summarize the characteristics of these cohesion and coupling metrics.

Table 1. Coupling metrics

Name	Definition
CBO [4][5][11]	Classes are coupled if methods or instance variables in one class are used by the other. CBO for a class is number of other classes coupled with it.
RFC [4][5]	Count of all methods in the class plus all methods called in other classes.
CF [3][6]	Classes are coupled if methods or instance variables in one class are used by the other. CF for a software system is number of coupled class pairs divided by total number of class pairs.
DAC[9]	The number of attributes having other classes as their types.

Table 2. Cohesion metrics

Name	Definition
LCOM [5]	Number of non-similar method pairs in a class of pairs.
LCOM3[7][9]	Number of connected components in graph whose vertices are methods and whose edges link similar methods.
RLCOM [10]	Ratio of number of non-similar method pairs to total number of method pairs in the class.
TCC [2]	Ratio of number of similar method pairs to total number of method pairs in the class.

All of these measures have two important features in common. First, they treat relationship between a pair of classes or methods as a binary quantity; second, they treat coupling and cohesion as an intransitive relation; that is no account is taken of the indirect coupling and cohesion, although two of cohesion (LCOM3 [7][9] and TCC [2]) have suggested extensions to incorporate indirect relationships between methods. In cohesion metrics, it should be noted that three of them (LCOM, LCOM3 and RLCOM) are in fact measures of lack of cohesion. TCC [2], in contrast to the other three metrics, measures cohesion rather than its absence. In other respects it is similar to RLCOM, being the number of similar method pairs divided by the total number of method pairs.

3. PROPOSED NEW METRICS

The study suggested that none of these measures was very effective in ranking the reusability of Java components. We therefore decided to develop alternative coupling and cohesion metrics in the hope of achieving superior performance. One obvious step was to develop measures that reflected the extent to which a pair of classes was coupled or a pair of methods resembled each other. Because none of the measures treated coupling or similarity as transitive relations, we decided that such indirect dependencies should be incorporated into our metrics.

3.1 Cohesion

We develop a cohesion metric that takes account of both the degree of cohesion and transitive (i.e indirect) cohesion between methods. Methods are said to be similar if the sets of instance variables that they access overlap. We adopt a graph theoretical approach. The methods of the class are the vertices. Suppose a class has a set of method members $M \equiv \{ M_1, M_2,\ldots M_m\}$ and let. $V_j \equiv \{V_{j,1}, \ V_{j,2}, \ \ldots. \ V_{j,n}\}$ be the instance variables accessed by method M_j. Then the edge from M_j to M_i exists if and only if $V_j \cap V_i$ is not null. Thus an edge of the graph reflects the similarity of the methods in that they have at least one instance variable in common. The similarity graph is undirected because intersection is a symmetric relation. The next step is to associate a number with each edge that reflects the extent to which the two methods have instance variables in common. We therefore define $SimD(i,j)$, our measure of direct similarity of two methods, M_i and M_j, as

$$SimD\ (i,\ j) = \frac{\left|V_i \cap V_j\right|}{\left|V_i \cup V_j\right|}$$

where $i \neq j$ ($SimD(j,j)$ is defined to be zero). Note that $1 \geq SimD(i,j) \geq 0$.

The extension of the measure to include indirect similarity proceeds along the same lines as we employed for indirect coupling. The strength of similarity provided by a path between two methods is the product of the SimD values of the edges that make up the path. Thus we define $SimT(i,j,\pi)$, the transitive similarity between methods M_i and M_j due to a specific path π, as

$$SimT\ \ (i,\ j,\pi\ ) = \prod_{e_{s,t} \in \pi} SimD\ \ (s,t) = \prod_{e_{s,t} \in \pi} \frac{\left|V_s \cap V_t\right|}{\left|V_s \cup V_t\right|}$$

where $e_{s,t}$ denotes the edge between vertices s and t. As in the case of coupling, the path with the highest SimT value is selected to define the similarity of the two methods, $Sim(i,j)$.

$$Sim\ \ (i,\ j) = SimT\ \ (i,\ j,\pi_{\max}\ )$$

where $\pi_{\max}\ (i,\ j) = \arg\ \max_{\pi \in \Pi} SimT\ \ (i,\ j,\pi\ )$ and Π is the set of all paths from M_i to M_j. This measure is used to provide a measure of the cohesion of the class, *ClassCoh*, by summing the similarities of all method pairs and dividing by the total number of such pairs:

$$ClassCoh\ \ = \frac{\sum_{i,j=1}^{m} Sim\ (i,\ j)}{m^2 - m}$$

where m is the number of methods in the class. Finally, the weighted transitive cohesion of the complete software system, *WTCoh*, is defined as the mean cohesion of all the classes of which it is comprised:

$$WTCoh\ \ = \frac{\sum_{j=1}^{n} ClassCoh_j}{n}$$

where n is the number of classes in the system.

3.2 Coupling

As with cohesion measure, we regard software system as a directed graph, in which the vertices are the classes comprising the system. Suppose such a system comprises a set of classes $C \equiv \{C_1, C_2,\ldots C_m\}$. Let $M_j \equiv \{M_{j,1}, \ M_{j,2}, \ \ldots. \ M_{j,n}\}$ be the methods of the class C_j, and $R_{j,i}$ the set of methods and instance variables in class C_i invoked by class C_j for $j \neq i$ ($R_{j,j}$ is defined to be null). Then the edge from C_j to C_i exists if and only if $R_{j,j}$ is not null. Thus an edge of the graph reflects the direct coupling of one class to another. The graph is directed since $R_{j,i}$ is not necessarily equal to Ri,j.

The next step is to associate a number with each edge that reflects the extent of direct coupling from one class to another. We define $CoupD(i,j)$, as the ratio of the number of methods in class j invoked by class I to the total number of methods in class I, which indicates the impact of class j to class i.

$$CoupD\ \ (i,\ j) = \frac{\left|R_{i,j}\right|}{\left|R_i\right| + \left|M_i\right|}$$

Then the indirect coupling between classes is included. Suppose that $CoupD(i,j)$ and $CoupD(j,k)$ have finite values but that $CoupD(i,k)$ is zero. Thus although there is no direct coupling between classes C_i and C_k, there is a dependency because C_i invokes methods in C_j which in turn invokes methods in C_k. The strength of this dependency depends on the two direct couplings of which it is composed, a reasonable measure is defined as:

CoupD(*i*,*j*) × *CoupD*(*j*,*k*). This notion is readily generalised. A coupling between two classes exists if there is a path from one to the other made up edges whose *CoupD* values are all non-zero. Thus we define *CoupT*(*i*,*j*,π), the transitive coupling between classes C_i and C_j due to a specific path π, as

$$CoupT\ (i, j, \pi) = \prod_{e_{s,t} \in \pi} CoupD\ (s,t) = \prod_{e_{s,t} \in \pi} \frac{|R_{s,t}|}{|R_s| + |M_s|}$$

$e_{s,t}$ denotes the edge between vertices *s* and *t*. Note first that CoupT includes the direct coupling, which corresponds to path of length one, and second that, because the CoupD values are necessarily less than one, transitive couplings due to longer paths will typically have lower values.

In general there may be more than one path having a non-zero CoupT value between any two classes. We simply select the path with largest CoupT value and hence define Coup(i,j), the strength of coupling between the two classes, C_i and C_j to be:

$$Coup\ (i, j) = CoupT\ (i, j, \pi_{\max})$$

where $\pi_{\max}(i, j) = \arg\max_{\pi \in \Pi} CPT\ (i, j, \pi)$ and Π is the set of all paths from C_i to C_j. The final step is to use measure between each pair of classes as a basis for a measure of the total coupling of a software system. The weighted transitive coupling (*WTCoup*) of a system is thus defined

$$WTCoup = \frac{\sum_{i,j=1}^{m} Coup\ (i, j)}{m^2 - m}$$

where *m* is the number of classes in the system.

4. AN EXPERIMENTAL COMPARISON

In our study, the metrics are used for a specific purpose: predicting how much effort would be required to reuse a component within a larger system. We therefore chose to measure reusability as simply the number of lines of code that were added, modified or deleted (NLOC) in order to extend its functionality in a prescribed way. The more lines required, the lower the reusability. This appears to us to be a crude but reasonable measure of the effort that would be required to adapt a component for use within a larger system. Three case studies were carried out: *Case 1 HTML Parser*: The original components analysed HTML documents, eliminated tags and comments and output the text. The required extension was to count and output the number of tags found during parsing.

Case 2 Lexical Tokenizer: The original components tokenized a text document using user supplied token rules and output the tokens on a web interface. The required extension was to count and output the number of tokens retrieved.

Case 3 Barcode: The original components accepted a sequence of alphanumeric characters and generated the corresponding barcode. The required extension was to count the number of letters.

For each case, 20 Java components were retrieved from a repository of about 10,000 Java components retrieved form the internet. The requisite extensions were then implemented by a very experienced Java programmer and NLOC counted. Despite the relative simplicity of the extensions, there was considerable variation in the quantity of extra code required. We then proceeded to investigate how successful the various measures of coupling and cohesion are in predicting this quantity. Our proposed metrics are compared with all the metrics reviewed in section 2. In order to present the results on the same graph, those measures that do not produce values in the range (0,1) (i.e. CBO, RFC, DAC, LCOM and LCOM3) were divided by 100.

5. RESULTS

Two approaches were used to evaluate the performance of the various measures in predicting reusability: linear regression and rank correlation.

5.1 Linear Regression

The regression lines obtained for the five cohesion measures when applied to the HTML parser components are shown in Figure 1. The results for the other two sets of components were similar. It is clear that some measures provide much more consistent predictors than others. There are no obvious systematic departures from linearity so the use of simple regression appears reasonable. The regression lines obtained for coupling measures demonstrate the same situation.

The coefficient of determination, R^2, provides a measure of how much of the variation in NLOC is accounted for by the measures. Table 3 and Table 4 display the values of R^2 obtained for each of the coupling and cohesion measures on all three sets of components. In each case, our proposed new measure, WTCoup and WTCoh gave the largest value of R^2, indicating that it was the best linear predictor of reusability. The remaining measures produced at least one R^2 value so low as to indicate that that the correlation was not significantly above chance at the 5% level.

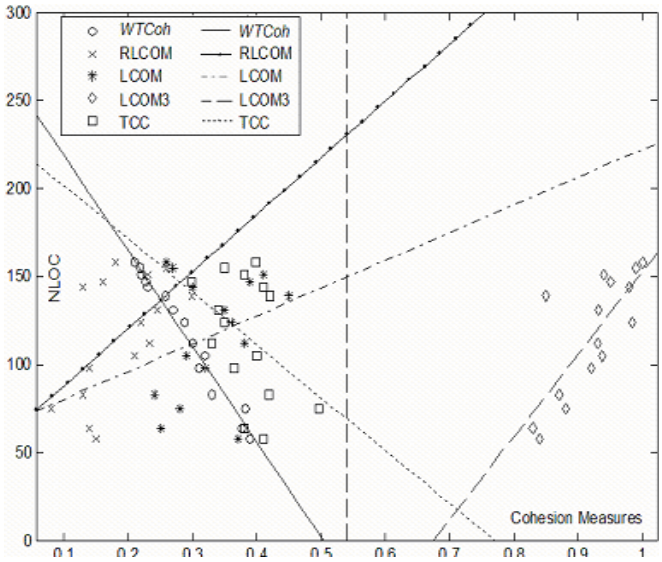


Figure 1. Regression of cohesion measures against reusability

Table 3. R^2 values for coupling measure regression lines.

Cases	WTCoup	CF	CBO	RFC	DAC
HTML Parser	.846	.621	.259	.793	.254
Lexical Token.	.836	.098	.004	.729	.738
Barcode Gen.	.958	.693	.121	534	.507

Table 4. R^2 values for cohesion measure regression lines.

Cases	WTCoh	RLCOM	LCOM3	LCOM	TCC
H. Parser	.847	.319	.259	.564	.178
L. Token.	.838	.783	.002	.709	.646
B. Gen.	.892	.702	.177	.101	.785

5.2 Spearman Rank Correlation

Although these results provide a strong indication that the proposed new measures are better predictors of reusability than the alternatives, our primary purpose is simply to rank a set of components retrieved from the repository. We therefore also computed the Spearman rank correlation coefficients between the rankings determined by NLOC and those produced by the various coupling and cohesion measures (Tables 5 and 6).

Table 5. Rank correlations values for coupling measures.

Cases	WTCoup	CF	CBO	RFC	DAC
HTML Parser	.975	.882	.465	.896	.507
Lexical Token.	.952	.291	.117	.822	.817
Barcode Gen.	.974	.758	.485	.656	.800

Table 6. Rank correlations values for cohesion measures.

Cases	WTCoh	RLCOM	LCOM3	LCOM	TCC
H. Parser	-.993	.522	.218	.564	-.343
L. Token.	.838	.783	.002	.709	.646
Bar. Gen.	.892	.702	.177	.101	.785

The relative performance of the various measures is consistent with the regression studies. In all cases, the two proposed measures, WTCoup and WTCoh, produced the highest rank correlations. They are in fact extremely high; no value was lower than 0.95.

6. DISCUSSION

These results clearly demonstrate that our proposed metrics for coupling and cohesion are very good predictors of the number of lines of code required to make simple modifications to Java components retrieved from the internet and are superior to other measures. The majority of coupling and cohesion metrics treat coupling and similarity as simple binary quantities and ignore the transitive relationship. Both our proposed measures concern these issues: First, they are weighted; that is, they use a numeric measure of the degree of coupling or similarity between entities rather than a binary quantity. Second they are transitive; that is, they include indirect coupling or similarity mediated by intervening entities. It is reasonable to enquire whether both these characteristics are necessary to achieve good prediction performance. In fact our investigations suggest that both contribute to the performance.

Although both WTCoup and WTCoh are good predictors, it is worth considering whether a linear combination might not produce even better results. Multiple regression for the Lexical Tokenizer components produced an R^2 of 0.981; the ranking produced using the regression coefficients to weight the terms had a Spearman correlation of 0.986. These are superior to the results produced by each metric alone but not by a great margin simply because there original results leave only modest scope for improvement. Developing such a composite quality measure would entail assuming the relative weighting of the two metrics should be the same for all types of component.

This work arose from, and is intended primarily as a contribution to, search engine technology. Nevertheless, we believe it may be of interest to a wider body of researchers: in particular, those involved in developing and evaluating software metrics.

7. ACKNOWLEDGMENTS

We are grateful to the four UK higher education funding bodies (for England, Scotland, Wales and Northern Ireland) for an Overseas Research Studentship (ORS/2002015010) awarded to G. Gui.

8. REFERENCES

[1] Gui, G. and Scott, P. D. Vector Space Based on Hierarchical Weighting: A Component Ranking Approach to Component Retrieval. In Proceedings of the 6th International Workshop on Advanced Parallel Processing Technologies (APPT'05)

[2] Bieman, J. M. and Kang, B-Y. Cohesion and Reuse in an Object-Oriented System. In *Proc. ACM Symposium on Software Reusability (SSR'95)*. (April 1995) 259-262.

[3] Briand, L., Devanbu, P. and Melo, W. An investigation into coupling measures for C++. *Proceedings of ICSE 1997*.

[4] Brito e Abreu, F. and Melo, W. Evaluating the impact of OO Design on Software Quality. *Proc. Third International Software Metrics Symposium.* (Berin 1996).

[5] Chidamber, S. R. and Kemerer, C. K. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, Vol. 20 (June 1994), 476-493.

[6] Harrison, R., S.J.Counsell, & R.V.Nith. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Transactions on Software Engineering*, Vol. 24 (June 1998), 491-496.

[7] Hitz, M. and Montazeri, B. Measuring coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing*. (Monterrey, Mexico, 1995).

[8] Kanmani, S., Uthariraj, R., Sankaranarayanan, V. and Thambidurai, P. Investigation into the Exploitation of Object-Oriented Features*. ACM Sigsoft, Software Engineering Notes*, Vol. 29 (March 2004).

[9] Li, W. & Henry, S. Object-Oriented metrics that predict maintainability. *Journal of Systems and Software*. 23(2) 1993 111-122.

[10] Li, X., Liu, Z. Pan, B. & Xing, B. A Measurement Tool for Object Oriented Software and Measurement Experiments with It. In *Proc. IWSM 2000*, 44-54.

[11] Subramanyam, R. & Krishnan, M. S. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Transactions on Software Engineering*, Vol. 29 (April 2003), 297-310.

TA-RE: An Exchange Language for Mining Software Repositories

Sunghun Kim[1], Thomas Zimmermann[2], Miryung Kim[3], Ahmed Hassan[4], Audris Mockus[5], Tudor Girba[6], Martin Pinzger[7], E. James Whitehead, Jr.[1], and Andreas Zeller[2]

[1]University of California, Santa Cruz, CA, USA
{hunkim, ejw}@cs.ucsc.edu

[2]Saarland University, Saarbrücken, Germany
{tz, zeller}@acm.org

[3]University of Washington, USA
miryung@cs.washington.edu

[4]University of Waterloo, Canada
aeehassa@plg.uwaterloo.ca

[5]Avaya labs
audris@avaya.com

[6]University of Berne, Switzerland
girba@iam.unibe.ch

[7]University of Zurich, Switzerland
pinzger@ifi.unizh.ch

ABSTRACT

Software repositories have been getting a lot of attention from researchers in recent years. In order to analyze software repositories, it is necessary to first extract raw data from the version control and problem tracking systems. This poses two challenges: (1) extraction requires a non-trivial effort, and (2) the results depend on the heuristics used during extraction. These challenges burden researchers that are new to the community and make it difficult to benchmark software repository mining since it is almost impossible to reproduce experiments done by another team. In this paper we present the TA-RE corpus. TA-RE collects extracted data from software repositories in order to build a collection of projects that will simplify extraction process. Additionally the collection can be used for benchmarking. As the first step we propose an exchange language capable of making sharing and reusing data as simple as possible.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms

Measurement, Experimentation

Keywords

Corpus, Software Repository Mining, Prediction, Analysis

1. INTRODUCTION

Software repositories, such as version archives, problem databases, newsgroups, and mailing lists, have been getting a lot of attention from researchers in recent years. They have been used to discover previously unknown information and evaluate existing software engineering approaches and theories. Mining software repositories (MSR) is an active research area.

MSR '06, May 22-23, 2006, Shanghai, China.

This has lead to a wide range of topics including co-change analysis [1, 3, 23], origin analysis [7, 11], signature change analysis [12], defect analysis and prediction [8], investigation of code clones [10], code decay [5], estimating drivers for software change effort [9] and quality [18], identifying key features of open source development process [14], chunking of software in order to facilitate distributed development teams [17], and constructing tools to identify expert developers [15].

Even though these research topics vary, every analysis needs to first extract data from software repositories. Developing such extraction tools requires a *non-trivial effort*, particularly for researchers new to this area. Kenyon was recently developed to simplify extraction from version archives [2]. However, such tools still require knowledge about version control systems and are thus difficult to learn.

Even though common tools may facilitate research, it remains *difficult to reproduce* existing results. First, some required information that is not available in software repositories has to be inferred using heuristics and through interviews of the project participants. The latter is often essential because different projects tend have different development processes and different change and reporting practices. Typical examples are the recovery of change transactions from CVS [22] and the identification of bug fixes [16]. The algorithms used differ widely in existing research efforts. Since choosing different parameters may lead to completely different results, *benchmarking is almost impossible*. Second, when analyzing open-source projects, researchers rely on the availability of those repositories in the future. However, this assumption is very optimistic in particular since many projects are currently migrating their CVS repositories to Subversion. As a result, the original CVS repositories may be gone in a few years.

We also want to analyze closed source projects. In the rare event such a code history becomes publicly available, it is unlikely we will have direct access to its SCM repository.

Other research areas address the above problems by providing a collection of common test cases or documents. Examples are the UCI Repository [19], the Reuters corpus [13] from text classification research, and the PROMISE Repository [20]. In this paper, we propose a similar solution: a collection of extracted software repositories called the TA-RE[1] corpus.

[1] TA-RE is a Korean word and means "group" or "cluster".

The TA-RE corpus consists of (1) an *exchange language* and (2) *extracted data* of a set of selected software projects that will allow researchers to reproduce and benchmark their experiments. The vision of TA-RE is that every paper on mining software repositories will share its extracted data via the TA-RE repository. Other researchers can then reuse this information without spending too much time on extraction.

TA-RE *needs to be widely accepted and adopted*; otherwise it will have no impact. One key to acceptance is for the data sharing to be as easy as possible. This leads to several requirements that are discussed in Section 2. The resulting exchange language is presented in Section 3, and Section 4 discusses alternatives to TA-RE and Section 5 presents related work. Section 6 concludes the paper with an outlook and future work.

2. REQUIREMENTS

The success of the TA-RE project depends on whether the research community will adopt it. Therefore we discuss several requirements for the corpus that would increase its appeal.

2.1 Completeness of Information

(E1) The exchange language should be able to describe all information that is available from most standard SCM systems:

1. *Transactions:* the author, date, log message, and the version of each changed file. This information enables reconstruction of proper snapshots.
2. *Changes:* the files that were changed, including their new content. This information suffices for lightweight syntactic analysis like creating abstract syntax trees.
3. *Snapshots:* a consistent state of a project after each transaction. This information is needed for static and dynamic program analysis, clone detection, etc.

Not all SCM systems provide the above information. For instance, for CVS the transaction information is not stored and has to be recovered by heuristics.

(E2) Additionally the exchange language should support information that can be inferred for most SCM systems:

1. *Source code positions* of classes, methods, or functions
2. *Size and location of the change:* which lines were added, deleted or modified
3. *Nature of a change:* adaptive, corrective, or perfective changes [16], fix-inducing changes [21]
4. *Counts:* number of methods, lines, changes or fixes
5. *References to other artifacts,* such as problem databases, mailing lists, and newsgroups

(E3) All information provided by the exchange language should come with a *quality* (or trust) annotation. A transaction from a Subversion archive may be of low quality if it was migrated from a CVS repository. Such annotations should describe known data quality problems or heuristics used to calculate the relevant attribute (see E4).

(E4) For inferred information the exchange language should provide ways to identify the *algorithm* that was used. Additionally, it should be possible to use different variants of an algorithm in the same dataset (e.g., different algorithms to recognize bug fixes).

(E5) The exchange language should be extensible in anticipation for new research interests.

2.2 Applicability to Research and Industry

(A1) The corpus should support closed-source projects. Such projects might be willing to share some information without revealing their actual source code. This means that TA-RE needs to provide tools to anonymize the extracted data. To simplify this process, the source code and the description of changes should be separated in the exchange language.

2.3 Usability

(U1) The exchange language should allow any researcher to provide new data with minimal effort.

(U2) The data from the corpus should be easy to use for researchers in their projects. In particular, the exchange language should be straightforward and must not be too difficult to parse, i.e., cross-references or complicated relations should be avoided.

(U3) The corpus itself should not be restricted to any platform. It should be usable for programs that are specific to any type of machine or system.

(U4) The exchange language should be well documented.

3. TA-RE CORPUS

We describe the TA-RE corpus exchange language in this section.

3.1 Available Information

The TA-RE exchange language can represent the following classes of data:

Extraction level 1: directly extractable data from SCM systems:
- Transaction information: author, transaction time, and change log
- All file contents (deltas) with the original directory structures
- File level co-change information

Extraction level 2: data obtained by further analysis, such as source code parsing “
- Entity (class, function, and method) level information and content
- File addition and deletion Information
- Unique identifier for each transaction and content
- Entity level co-change information

Mined data: data extracted using heuristics:
- Recovered transactions (CVS [22])
- Transaction, file, and entity level bug-fix data [6]
- Fix-inducing data at file and entity levels [21]
- Accumulated bug count at file and entity levels
- Origin relationship between entities [7, 11]
- Reference links among transactions, contents, and entities.

3.2 Corpus Model

The TA-RE corpus data contains two flavors: transaction and content data. The corpus data has multiple transactions, and a transaction has multiple contents. Instead of providing all contents of each transaction, TA-RE provides only changed (added, deleted, and modified) contents in each transaction, since it is possible to recover all transaction contents from only the changed contents. The content data consist of two parts: content metadata and original file content. The content metadata has metadata for the original file content such as reference, change status, count, and entity information. We separate the content metadata and original file content for two reasons: (1) to store binary files and (2) to make original file contents optional for closed source projects.

We use sequential numbers (starting from 1) for transaction and content identifiers. The transaction identifiers are ordered chronologically, hence the transaction 1 is older than the transaction 2. We can easily determine the transaction order from transaction identifiers. The content identifier is unique for the same file name. For example, file '*/src/foo.java*' will have the same identifier over all transactions.

Since we use numeric identifiers for both transactions and contents, we use prefixes to avoid possible confusion between their identifiers. We use the prefix '*t*' for transactions and the '*c*' for contents. Content metadata is stored as a file whose name is the combination of the content prefix and a content identifier such as *'c32'*. Since content exchange language consists of metadata data and original file content, we use file extensions to distinguish them: '*.meta*' for the metadata and '*.con*' for the original file content.

Each transaction has a directory whose name is the combination of the transaction prefix and a transaction identifier. Transaction information is stored as a file, '*transaction*' in the corresponding transaction directory. All contents (*.meta and *.con) of the transaction are stored in the directory as well. For example, for transaction 1, the '*t1*' directory is created, and contains the transaction information file ('*transaction'*) and content files ('*c[content-id].meta'* and '*c[content-id].con'*) of the transaction.

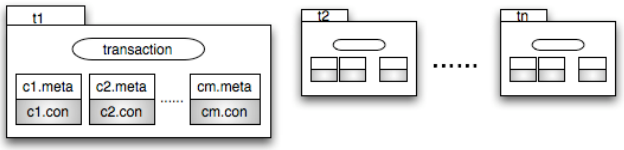


Figure 1. TA-RE Corpus Model

Figure 1 shows the TA-RE corpus model. Each transaction directory ('*t[transaction-id]'*) has three kinds of corpus files:

> **Transaction information** ('*transaction*')**:** information of the corresponding transaction.
> **Content metadata** ('*c[content-id].meta'*)**:** metadata of the content
> **Original file content** ('*c[content-id].con*'): the original file content (optional)

The transaction and content metadata exchange language are formatted using XML. An example of transaction corpus exchange language is shown in Figure 2. It has the TA-RE exchange language version number, transaction id, release, author, data, indication of transaction nature, and change logs.

```
<?xml version="1.0" encoding="utf-8" ?>
<T:transaction xmlns:T="TA-RE:" id="t32">
  <T:corpus-version>0.1</T:corpus-version>
  <T:author>hunkim</T:author>
  <T:date>1995.3.1.1 xxx GMT</T:date>
  <T:nature kind="release" value="release 1.0"/>
  <T:nature kind="fix" heuristic="mockus2000"/>
  <T:nature kind="fix" heuristic="fischer2003"/>
  <T:change-log>Fixed compilation error in foo.c
  </T:change-log>
</T:transaction>
```

Figure 2. An example of transaction data

Figure 3 shows an exchange language example of a content metadata file. Only changed content data (added, deleted, modified) are present in TA-RE. The metadata have the original file name, references, counts, and entity data. The original file content can be fount at the '*c[content-id].con*' file in the same transaction directory. The detailed XML elements and DTD are described in http://tare.dforge.cse.ucsc.edu/.

```
<?xml version="1.0" encoding="utf-8" ?>
<T:content xmlns:T="TA-RE:" id="c32"
           filename="src/edu/ucsc/Kenyon.java">
  <T:corpus-version>0.1</T:corpus-version>
  <T:change-status value="modified"/>
  <T:reference kind="partof" level="transaction"
       transaction-id="t40"/>
  <T:reference kind="fixes" level="content"
       transaction-id="t29" content-id="c32"/>
  <T:reference kind="fixed-by" level="content"
       transaction-id="t45" content-id="c32"/>
  <T:reference kind="fixed-by" level="content"
       transaction-id="t99" content-id="c32"/>
  <T:count kind="accumulated-fix" value="2"/>
  <T:count kind="accumulated-fix-inducing" value="3"/>
  <T:count kind="accumulated-change" value="10"/>
  <T:entity level="class" id="class-foo" name="Foo"
      start-pos="20" end-pos="2564">
  <T:entity level="method" id=" foo"  name="foo"
       return-type="void"   parameters="int I, char *var"
       start-pos="32" end-pos="95">
      <T:reference kind="fixes" level="entity">
       tansaction-id="t23" content-id="c32" entity-id="foo"/>
  </T:entity>
  <T:entity level="method" id=bar" name="bar"
       return-type="char"   parameters="int i, char c"
       start-pos="103" end-pos="195">
  </T:entity>
     ...
  </T:entity>
</T:content>
```

Figure 3. An example of a content metadata file. This content fixes the same content at transaction 29. This change includes bugs (fix-inducing changes). The bugs in this content change are fixed in transaction 45 and transaction 99. The original file content is stored in '*c32.con*' in the '*t40*' directory.

4. DISCUSSION

4.1 Why not use Traditional Extractors?

There are SCM fact extractors such as Kenyon [2] and APFEL [4]. These extractors are useful for extracting data from SCM systems without dealing with the SCM connections or protocols directly. Choosing different extractor options will yield different data from the same SCM repository. For example, the number of transactions and the number change contents of a transaction may be different when extraction tools use different CVS sliding windows times. Mined data in TA-RE such as bug-fix data or origin analysis data need to be provided by the extractor using their own heuristic options. Extracting different data from the same SCM systems makes it *difficult to reproduce* existing results.

4.2 Why not use DBMS Schemas?

Fischer et al. proposed DBMS schemas [6] to store data for software repository mining research. If the schema is complete and publicly available, the data in DBMS are beneficial for all software repository mining researchers. TA-RE provides an exchange language. It does not enforce any universal database schema because different research might need different formats. Use of an exchange format avoids this issue, as each researcher can write tools to export TA-RE to their project specific DB schema. Every researcher only has to write the import/export tools once and can reuse them for every project she downloads from TA-RE

4.3 Why not use Transaction-Aware SCM?

Transaction-aware SCM systems such as Subversion provide change based revision numbers (no need to recover transactions), log renaming events, and support metadata-setting features. Using SCM systems requires an extraction process, and it has the same limitations of using extractors (Section 4.1). TA-RE provides downloadable and ready-to-use data including all mined data such as bug-fix and big-inducing change information, which are not provided by SCM systems such as Subversion.

4.4 Closed Source Project Support

The TA-RE corpus exchange language can be used for closed source projects. First author information in transaction data files can be replaced with numeric ids to hide real author ids. The file names in the content metadata file can be omitted or replaced with obfuscated names. The original file contents stored in separate files (*c[content-id].con*) can be omitted. In addition, all entity information can be omitted.

5. RELATED WORK

The PROMISE repository provides various data sets for predictive model research in software engineering [20]. Data sets in the PROMISE repository mostly consist of features and classes or values. Using the features, researchers develop prediction models to predict classes (classification) or values (regression). PROMISE data sets are limited for general software repository mining research. Since they provide pre-defined features such LOC, count of operators, and count of blank lines, it is hard to extract new features that are not defined in the data set. The data sets are focused on developing predictive models. The non-predictive model research such as origin analysis, code clone genealogy, or co-change analysis cannot be performed using the data sets in PROMISE repository.

The UCI Repository of machine learning [19] or Reuters Corpus [13] are de facto standard benchmarking data set for text classification research. The data stets enables researchers to compare their classification results with others. TA-RE is inspired from them, but their data sets are designed for the text classification.

6. CONCLUSION AND FUTURE WORK

It is no secret that the majority of time spent during software repository mining is focused on extracting data. Additionally, the "magic" that is involved in the extracting phase makes comparison of results and benchmarking impossible. The TA-RE project addresses this issue by specifying a common exchange language that will be used to share project data. Using a common exchange language will enable reuse of data as much as possible. The next steps of this project are the following:

Finalize exchange language. This paper serves as a proposal for a common exchange language. Thus, designing a common language will heavily benefit from discussions and participation of other researchers. We hope that the discussions at the MSR workshop will give us enough feedback to finalize the exchange language.

Provide initial dataset. Once the exchange language is finalized, the participants of the TA-RE project will create an initial dataset for several selected projects.

Include other data sources. The initial exchange language will describe data only from version archives. For the next release, we plan to include additional data sources such as problem databases, mailing lists, or newsgroups.

For more information visit: http://tare.dforge.cse.ucsc.edu/
or join the discussion: http://groups.google.com/group/TaRe

7. REFERENCES

[1] J. Bevan and E. J. Whitehead, Jr., "Identification of Software Instabilities," Proc. of 2003 Working Conference on Reverse Engineering (WCRE 2003), Victoria, Canada, 2003.

[2] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[3] D. Beyer and A. Noack, "Clustering Software Artifacts Based on Frequent Common Changes," Proc. of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005), St. Louis, Missouri, USA, pp. 259-268, 2005.

[4] V. Dallmeier, P. Weißgerber, and T. Zimmermann, "APFEL: A Preprocessing Framework For Eclipse," 2005, http://www.st.cs.uni-sb.de/softevo/apfel/.

[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering*, vol. 27, pp. 1-12., 2001.

[6] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of 2003 Int'l Conference on Software Maintenance (ICSM'03), pp. 23-32, 2003.

[7] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. on Software Engineering*, vol. 31, pp. 166- 181, 2005.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, pp. 653-661, 2000.

[9] T. L. Graves and A. Mockus, "Inferring Change Effort from Configuration Management Data," Proc. of In Metrics 98: Fifth International Symposium on Software Metrics, Bethesda, Maryland, pp. 267-273, 1998.

[10] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 187-196, 2005.

[11] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships," Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005), Pennsylvania, USA, 2005.

[12] S. Kim, E. J. Whitehead, Jr., and J. Bevan, "Analysis of Signature Change Patterns," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 64-68, 2005.

[13] D. Lewis, Y. Yang, T. Rose, and F. Li, "RCV1: A New Benchmark Collection for Text Categorization Research " *Journal of Machine Learning Research*, vol. 5, pp. 361-397, 2004.

[14] A. Mockus, R. F. Fielding, and J. Herbsleb, "A Case Study of Open Source Development: The Apache Server," Proc. of 22nd Int'l Conference on Software Engineering (ICSE 2000), Limerick, Ireland, pp. 263-272 2000.

[15] A. Mockus and J. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," Proc. of 24rd Int'l Conference on Software Engineering (ICSE 2002), Orlando, Florida, pp. 503-512, 2002.

[16] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," Proc. of International Conference on Software Maintenance (ICSM 2000), San Jose, California, USA, pp. 120-130, 2000.

[17] A. Mockus and D. M. Weiss, "Globalization by Chunking: a Quantitative Approach," *IEEE Software*, vol. 18, pp. 30-37, 2001.

[18] A. Mockus, P. Zhang, and P. Li, "Drivers for Customer Perceived Software Quality," Proc. of 2005 Int'l Conference on Software Engineering (ICSE 2005), Saint Louis, Missouri, USA, 2005.

[19] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz, "UCI Repository of machine learning databases," 1988, http://www.ics.uci.edu/~mlearn/MLRepository.html.

[20] J. Sayyad Shirabad and T. J. Menzies, "The PROMISE Repository of Software Engineering Databases," 2005, http://promise.site.uottawa.ca/SERepository.

[21] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 24-28, 2005.

[22] T. Zimmermann and P. Weißgerber, "Preprocessing CVS Data for Fine-Grained Analysis," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2004), Edinburgh, Scotland, pp. 2-6, 2004.

[23] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Engineering*, vol. 31, pp. 429-445, 2005.

The Evolution Radar: Visualizing Integrated Logical Coupling Information

Marco D'Ambros, Michele Lanza, Mircea Lungu
Faculty of Informatics
University of Lugano, Switzerland
{marco.dambros, michele.lanza, mircea.lungu}@lu.unisi.ch

ABSTRACT

In software evolution research logical coupling has extensively been used to recover the hidden dependencies between source code artifacts. They would otherwise go lost because of the file-based nature of current versioning systems. Previous research has dealt with low-level couplings between files, leading to an explosion of data to be analyzed, or has abstracted the logical couplings to module level, leading to a loss of detailed information. In this paper we propose a visualization-based approach which integrates both file-level and module-level logical coupling information. This not only facilitates an in-depth analysis of the logical couplings at all granularity levels, it also leads to a precise characterization of the system modules in terms of their logical coupling dependencies.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance, Version Control, Reengineering, Reverse Engineering

General Terms

Measurements, Design

Keywords

Evolution, Logical Coupling, Visualization

1. INTRODUCTION

Versioning systems allow developers to record the history of a software project. The facilities given by versioning systems and the amount of data retrieved fostered the research field of software evolution [13], whose goal is to analyze the history of a software system and infer causes of its current problems, and possibly predict its future.

The history of a software system also holds information about the logical couplings. These are implicit and evolutionary dependency relationships between the artifacts of a system which, although potentially not structurally related, evolve together and are therefore linked to each other from a development process point of view. In short, logically coupled entities have changed together in the past and are thus likely to change in the future. Logical coupling information can therefore be used to predict the evolution of a software system. Moreover, logical coupling information reveals potentially misplaced artifacts in a software system, because entities that evolve together should be placed close to each other for cognitive reasons: A developer who modifies a file in a system could forget to modify related files because they are placed in other subsystems or packages.

In this paper we propose a technique to inspect logical coupling relationships, which integrates information both at a module-level (which subsystems are coupled with each other) and at a file-level (which files are responsible for the logical couplings). Our technique is based on a specific visualization that we named *Evolution Radar*. Visualization techniques have already been successfully used to study the evolution of software systems [1, 5, 10, 11, 14, 16, 17].

With our approach we tackle the following problems:

- How to present very large amounts of evolutionary information in an effective way.
- How to render logical coupling relationships in an intuitive way.
- How to enable a developer to study and inspect these relationships and to guide him to the files that are responsible for the logical couplings.

All the results and the examples presented in the Paper have been obtained by applying the presented visualization technique on the Mozilla (www.mozilla.org) case study.

Structure of the paper. In Section 2 we discuss the research that has been performed on logical coupling. In Section 3 we introduce our approach based on the *Evolution Radar* to render logical coupling information. We validate our technique on a large software system in Section 4 and we look at related work in Section 5. In Section 6 we conclude the Paper by summarizing our contributions and give an outlook on our future work in this field.

2. LOGICAL COUPLING

Logical coupling represents the implicit dependency relationship between two or more software artifacts that have been observed to frequently change together during the evolution of the system. This co-change information can either be present in the versioning system, or must be inferred by analysis. For example subversion marks co-changing files at commit time as belonging to the same *change set* while the files which are logically coupled must be inferred from the modification time of each individual file.

MSR'06, May 22–23, 2006, Shanghai, China.

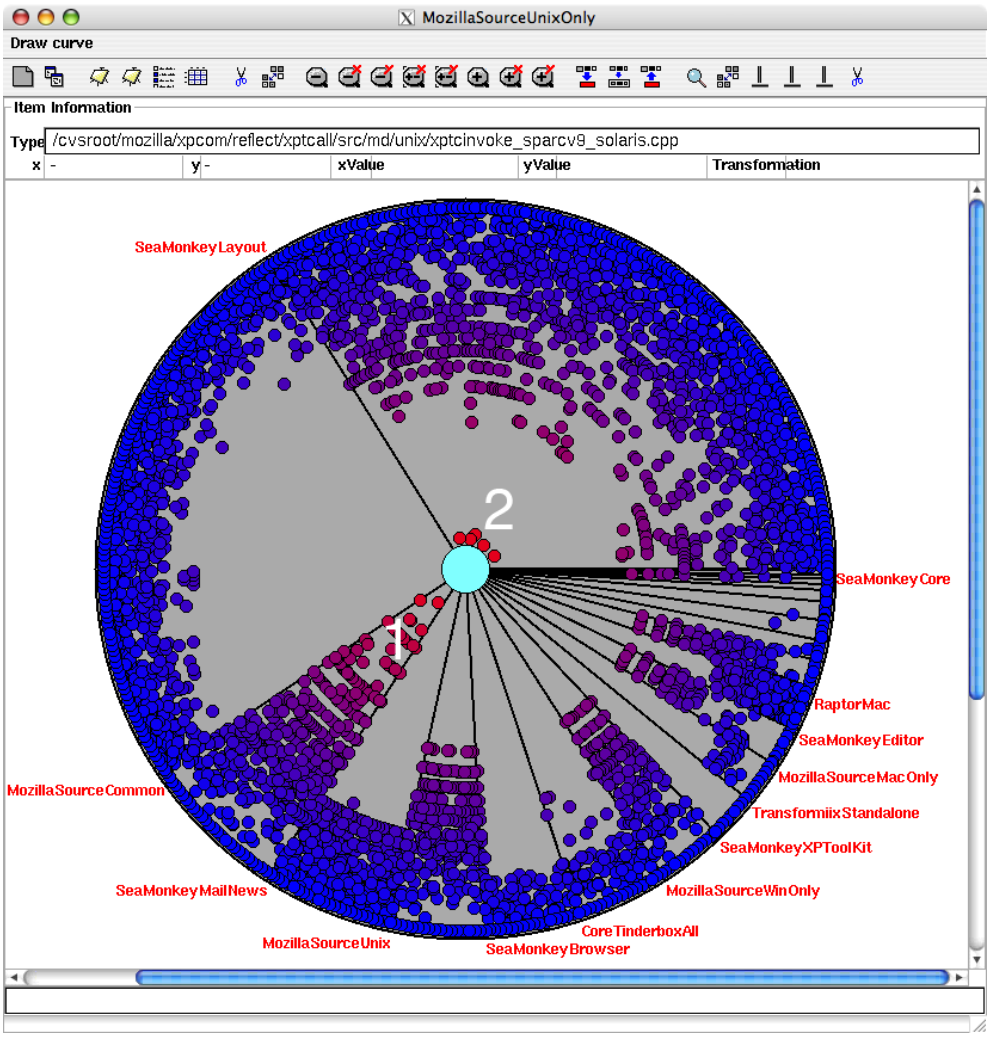


Figure 1: A sample Evolution Radar visualization of the core Mozilla modules.

The concept of logical coupling was first introduced by Gall *et al.* [7] to detect implicit relationships between modules. The technique that they proposed uses information from the CVS versioning control system to detect dependencies between the modules of a system. They used logical coupling to analyze the dependencies between the different modules of a large telecommunications software system and show that the approach can be used to derive useful insights on the architecture of the system. There are two reasons why the technique proved to be useful:

- It is more lightweight than structural analysis, as it needs to analyze a smaller amount of data, *i.e.*, only the data provided by the CVS log files. Moreover, as it works at text level, it can analyze systems written in multiple languages without the trouble of parsing and analyzing the data.
- It can reveal dependencies that are not structural, and therefore are not present in the code or in the documentation. These dependencies are the most troublesome and are prone to represent sources of bugs in software projects.

Later the same authors revisited the technique to work at a lower abstraction level. They detected logical couplings at class level [8] and validated it on 28 releases of an industrial software system. The authors showed through a case study that architectural weaknesses such as poorly designed interfaces and inheritance hierarchies could be detected based on logical coupling information.

Ratzinger *et al.* [15] used the same technique for analyzing the logical coupling at the class level with the aim of learning about, and improving the quality of the system. To accomplish this, they defined *code smells* based on the logical coupling between classes of the system.

Working at a finer granularity level, Zimmermann *et al.* [20] used the information about changes that are occurring together to predict entities that are likely to be modified when one is being modified.

The main problem with the mentioned approaches is that they either work at the architecture level, *i.e.*, without knowing which finer-grained entities cause the logical coupling, or they work at the file (or even finer) granularity level, *i.e.*, losing the global view of the system.

In this paper we propose an approach to overcome this shortcoming by means of a visualization technique called *Evolution Radar*, presented next.

3. THE EVOLUTION RADAR

The Evolution Radar is a visualization technique to render file-level and module-level logical coupling information in an integrated and interactive way, thus allowing the viewer to navigate and query the visualized data. It is implemented in BugCrawler [6].

3.1 Principles

The Evolution Radar (see Figure 1) visualizes the logical coupling of one module with the others. The module in focus is placed in the middle of a pie chart, where each sector represents one of the other modules. The size of each sector depicts the size of each module in terms of number of files. The modules are sorted according to this size metric.

The files of each of those modules are represented as colored circles and placed according to the logical coupling they have with the module placed in the center (the closer the files are to the center, the more coupled they are). The logical coupling can also be mapped on the color of the figures using a temperature representation: the hotter (from blue to red) the color is, the higher the value of the logical coupling measure is.

In the Evolution Radar it is possible to use different time interval combinations for the computation of the logical coupling and any combination of the position-color mapping. For example we can use the last year as time interval and map the resulting measure on the position, while for the color we compute the logical coupling considering the last month of the history of files.

3.2 Example

In the example Evolution Radar depicted in Figure 1 we see that one module (MozillaSourceCommon marked as 1) in the lower half of the radar has many files which are strongly coupled with the files in the center module (MozillaSourceUnixOnly). As a result we can say that these modules are strongly logically coupled.

Moreover, we see that the largest module (SeaMonkeyCore marked as 2) contains many files (the sector is big), but only a few are logically coupled with the center module. These files should be investigated to see whether they should be moved to the center module.

3.3 Logical Coupling Measure

We define the logical coupling between a file *f* and a module *M* as the maximum logical coupling between *f* and all the files belonging to *M*. The coupling between two files is defined as the number of "shared commits" they have, *i.e.,* the commits performed within a fixed time window[1]. This approach can be improved using a sliding time window as proposed by Zimmermann *et al.* in [19]. Using a sliding time window the obtained set of shared commits is a superset of the one obtained using a fixed time window. Thus the results found with our approach are still valid with the sliding time window, but with the latter it might be possible to find more of them.

To avoid outliers (files with a very high value of logical coupling with respect to the average) to deform our visualization, *i.e.,* pushing all the other figures to the boundary, we use a percentage value. We divide the number of shared commits by the average of the total number of commits of the two files. However, the percentage measure does not weigh the logical coupling with the absolute number of commits, implying that a file with 5 commits has the same value of a file with 100 commits if they have the same number of shared commits. A solution to this problem consists in multiplying the logical coupling percentage value with the logarithm of the total number of commits. Experiments with both the measures show that even with the log scale some entities are displaced too much from the center because of the outliers. Thus we choose the percentage value as logical coupling measure, solving the problem using a simple query engine. It allows the user to select and/or remove from the view all files having a number of commits below a given value.

[1]We use a 200 seconds time window, as used in [19].

3.4 Advantages

The Evolution Radar is interactive, *i.e.,* the user can zoom in on details, can select, inspect, remove single files, *etc.* to verify hypotheses such as whether certain files should be moved from one module to the other.

The Evolution Radar has several advantages regarding its visual expressiveness: It is rotation invariant like Chuah's time wheel visualizations [4]. It occupies a settable amount of screen space, *i.e.,* it is always possible to visualize the whole radar on screen, independent of its resolution. It does not visualize the coupling relationships as edges and therefore does not suffer from overplotting: The radar always remains intelligible, *i.e.,* it is easy to make out the heavily coupled modules which are displayed as "spikes" pointing to the center. It is also easy to make out single files responsible for the coupling which are placed close to the center. The Evolution Radar is applicable not only to modules, but also to any set of files.

4. MOZILLA EXPERIMENTS

The Evolution Radar helps in answering questions about the evolution of a system which are useful to developers, analysts, and project managers:

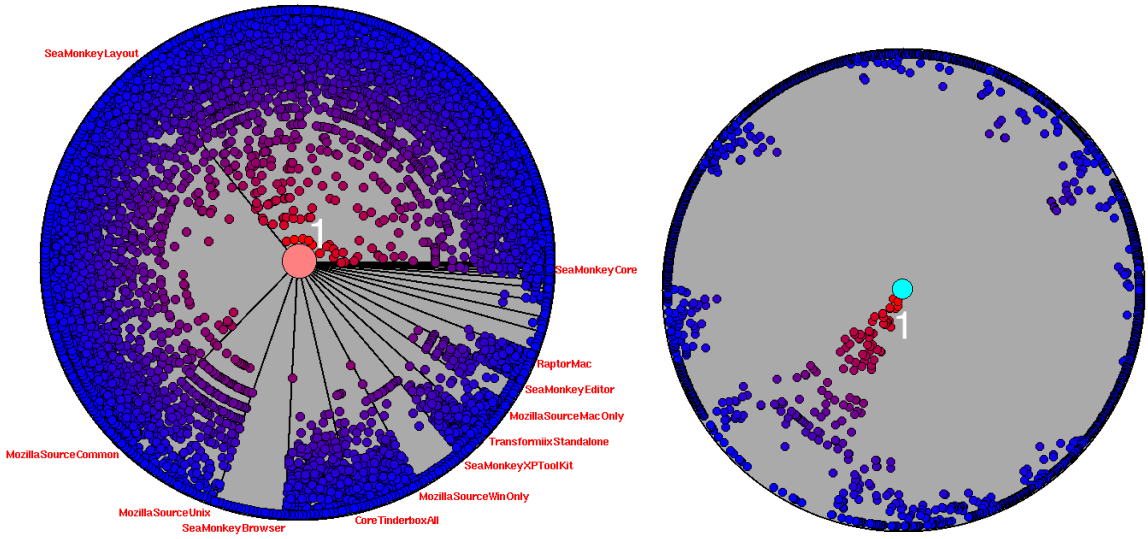
- *Developers* can use the technique to answer the question: "If I change this file, what others will I have to modify?". The Evolution Radar offers a visual way to assess the files that might change in the future based on the prediction offered by logical coupling. Due to the fine-grained level of the visualization, files can be inspected individually.

- *Analysts and project managers* can use the Evolution Radar to (i) understand the overall structure of the system in terms of module dependencies, (ii) examine the structure of these dependencies at the file granularity level and (iii) get an insight of the impact of changes on a module over other modules. This knowledge will help them in (i) localizing where refactorings should be applied, (ii) deciding whether certain files should be moved to other modules, and (iii) understanding the evolution of the logical coupling among modules.

In the remainder of this section we provide example scenarios of applying the Evolution Radar technique on 30'000 source code files in the Mozilla case study. For each example we mention which was the goal of the analysis and the potential stakeholders. Throughout the examples the color metric is the same as the distance metric unless otherwise specified.

4.1 Understanding SeaMonkeyMailNews

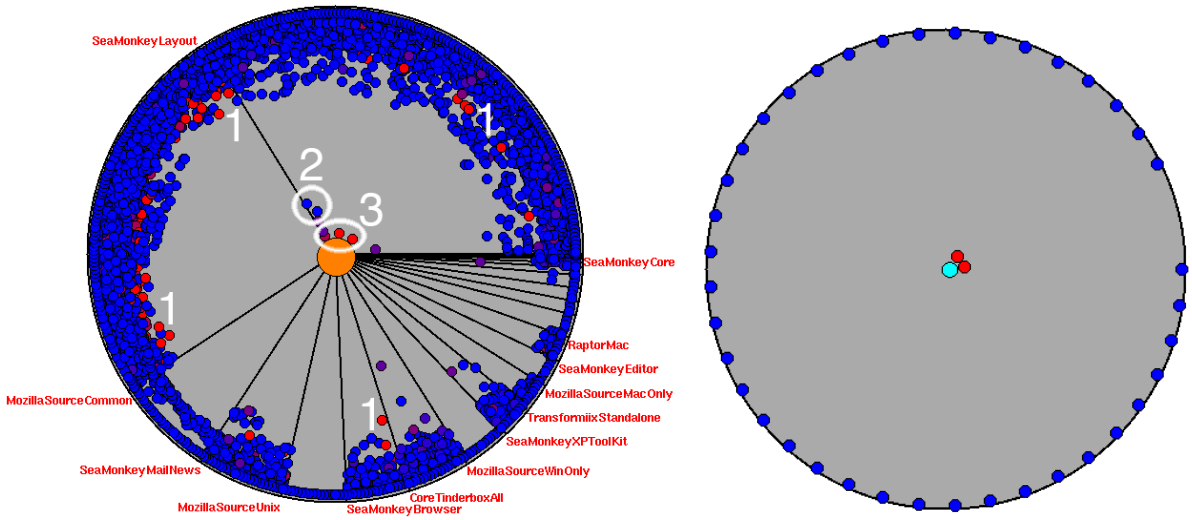
Target Audience: Analysts, project managers.
Goals. (1) To understand the dependencies between a module and all the other modules, (2) to understand the causes of these dependencies, and (3) to get an insight on the impact of changes regarding the target module.
Analysis. SeaMonkeyMailNews is a large module (1302 files) with strong dependencies with all the other modules in the system, especially with the largest module SeaMonkeyCore (7834 files). Figure 2(a) shows the Evolution Radar of SeaMonkeyMailNews. Many files are involved in the logical coupling between SeaMonkeyCore and SeaMonkeyMailNews. This information is useful but



(a) The Evolution Radar of the SeaMonkeyMailNews module.

(b) The details of the logical coupling between SeaMonkeyMailNews and SeaMonkeyCore.

Figure 2: Evolution Radars for SeaMonkeyMailNews.



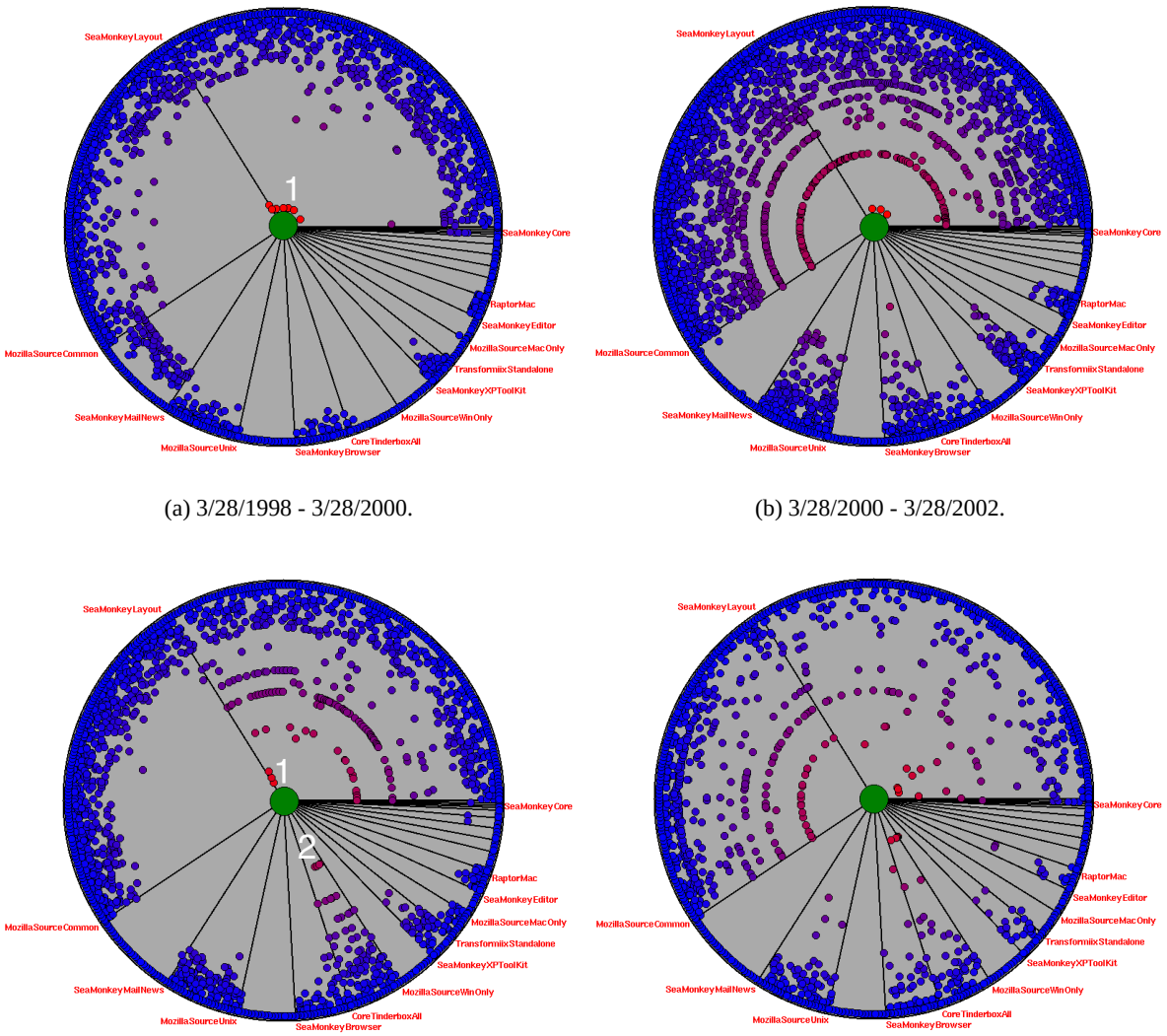
(a) The Evolution Radar of the ThunderbirdTinderbox module.

(b) The details of the logical coupling between ThunderbirdTinderbox and the `rdf/chrome/src/*` files.

Figure 3: Evolution Radars for ThunderbirdTinderbox.

still too coarse-grained. Thus we need to understand how the logical coupling is structured in terms of the individual files.

We refine the view of the logical coupling between modules by selecting the files closest to the center that are marked as (1) in Figure 2(a), and reapply the Evolution Radar for them. Now the group of selected files belonging to SeaMonkeyCore plays the role of the module in the center (represented as the cyan disc in Figure 2(b)), and the contents of SeaMonkeyMailNews, which was the previous center module, are scattered around them. As we can see from Figure 2(b) the logical coupling is due to the files marked as 1. All these files belong to the `mailnews/db/mork` directories tree, while the ones marked as 1 in Figure 2(a) belong to `db/mork`. These two hierarchies should be further inspected and, in case, merged and moved to the appropriate module.



(a) 3/28/1998 - 3/28/2000.

(b) 3/28/2000 - 3/28/2002.

(c) 3/28/2002 - 3/28/2004.

(d) 3/28/2004 - 1/9/2005.

Figure 4: The Evolution of the PhoenixTinderbox logical couplings.

4.2 ThunderbirdTinderbox Impact Analysis

Target Audience. Developers.
Goals. (1) To understand the change impact at the file level (*i.e.*, answer the question: "If a file changes in this module, what other files might have to change?").
Analysis. ThunderbirdTinderbox is a small module (42 files) but it has dependencies with most of the other Mozilla modules. As developers, we want the best candidates, *i.e.*, the files which have not only the strongest logical couplings but also the most recent ones. We tackle the problem by creating an Evolution Radar view which presents two types of logical coupling: (i) Coupling computed for the whole history of the system and (ii) coupling computed for the last 6 months only.

To present both types of information in the same figure we map the logical coupling based on the history of the last 6 months on the color of the discs representing files, while keeping the distance to represent the logical coupling for the whole history. In Figure 3(a) we can see three types of files:

- The files in group (3) are the most interesting, as they were always, and especially during the last 6 months, changed together with some files of the ThunderbirdTinderbox module.
- For the files marked with (1), the logical coupling is weak when computed for the whole history, but is strong when computed considering only recent changes.
- For the files in group (2), the inverse holds, because the coupling recently decreased (strong for the whole history, weak for the last six months).

To continue the analysis, we focus our attention on the files in group (3) only (all of which belong to the `rdf/chrome/src/` directory), because we want our candidate set to be small. We build another Evolution Radar (Figure 3(b)) using the set of files

in group (3) as the reference point and with the files belonging to the ThunderbirdTinderbox module scattered around it.

We find out that the logical coupling is due to two files only (`chrome/src/nsChromeURL.cpp` and `.h`) in the ThunderbirdTinderbox module. This means that if we want to modify the `nsChromeURL` files, it is very likely that we have to modify the files belonging to group 3 as well, while for all the other ThunderbirdTinderbox files we don't have this problem.

4.3 The Evolution of PhoenixTinderbox

Target Audience. Project managers, analysts.
Goals. (1) To obtain a high-level insight about the past evolution of the logical coupling relationships of a certain module during development phases, and (2) to understand whether the logical coupling relationship of a module is "ameliorating" or "degrading". It is degrading if the module is more and more logically coupled to the others, leading to maintenance problems and suggesting refactoring.
Analysis. The evolution of the logical coupling for the module PhoenixTinderbox (depicted in Figure 4) shows that the module went through diverse phases.

1. In the first phase from 1998 to 2000 it was decoupled from most of the system modules except SeaMonkeyCore because of the few files marked as (1).

2. Between 2000 and 2002 its architecture degraded since it became more coupled with other modules, namely: SeaMonkeyLayout, SeaMonkeyMailNews, SeaMonkeyBrowser, SeaMonkeyXPToolKit.

3. Between 2002 and 2004, possibly due to a restructuring phase, most of the logical couplings were reduced but the co-dependency with SeaMonkeyCore remained (marked as 1) and the one with CoreTinderboxAll increased (marked as 2).

4. In the last phase the architecture degraded again since (i) the dependencies with SeaMonkeyCore were reduced but still remained, (ii) the logical coupling with SeaMonkeyLayout became strong again and (iii) the dependencies with SeaMonkeyMailNews, CoreTinderboxAll and SeaMonkeyEditor were slightly increased.

5. RELATED WORK

Since Section 2 already introduced related work on logical coupling, this section presents work related to software evolution visualization.

A similar approach to visualize logical coupling has been presented by Pinzger et al. [14] with Kiviat Diagrams. As a difference they do not visualize file-level information but use surfaces to depict complete releases, while in our visualization we depict all evolving files in one diagram. Another difference is that they represent the coupling as edges between the visible modules.

The graph based representation in which entities involved in logical coupling were nodes in a graph and coupling was represented as edges between them was used since the first publications related to logical coupling [7, 8]. However, the problem with this representation is that it either represents only modules, and then it is too coarse grained, or it represents modules and files, but then it does not scale to large systems.

A visual data-mining tool to represent both binary association rules and n-ary association rules is EPOsee [3]. The tool adapts standard visualization techniques for association rules to also display hierarchical information.

Chuah and Eick present a way to visualize project information through glyphs called infobugs. Glyphs are graphical objects representing data through visual parameters. Their infobug glyph's parts represent data about software [4]. The difference with respect to our work is that they use glyphs to view project management data, while our work focuses on describing how a module is logically coupled to the others. One common advantage is that both approaches are rotation invariant.

Lanza's Evolution Matrix [12] visualizes the system's history in a matrix in which each row is the history of a class. A cell in the Evolution Matrix represents a class and the dimensions of the cell are given by evolutionary measurements computed on subsequent versions. The evolution matrix does not represent any relationship between the evolving entities.

Bayer [2] computes a co-change graph and proposes a layout which reveals clusters of frequently co-changed artifacts. Jazayeri *et al.* [11] visualizes software release histories using colors and the third dimension. They do no visualize any coupling relationships between modules.

Girba *et al.* used the notion of history to analyze how changes appear in the software systems [9] and succeeded in visualizing the histories of evolving class hierarchies [10].

Taylor and Munro [16] visualized CVS data with a technique called *revision towers*. Ball and Eick [1] developed visualizations for showing changes that appear in the source code.

Rysselberghe and Demeyer used a simple visualization based on information in version control systems to provide an overview of the evolution of systems [17].

Wu *et al.* described an Evolution Spectrograph [18] that visualizes historical sequences of software releases.

6. CONCLUSION

In this paper we have presented the Evolution Radar, a novel approach to integrate and visualize module-level and file-level logical coupling information. Unlike the previous visualizations in this domain, our approach facilitates an in-depth analysis of logical coupling between entities at different granularity levels. The visualization is useful to answer questions about the evolution of the system, the impact of changes at different levels of abstraction and the need for system restructuring.

We have provided solutions for the problems mentioned in Section 1: The Evolution Radar presents large amounts of information in a condensed way (in the Mozilla examples the number of files was greater than 30'000), guiding the user directly to the files responsible for the modules' logical coupling. The interactive facilities provided by our tool allow the user to inspect/filter entities of interest, to group them and to create ad-hoc visualizations on the fly.

As a case study, we have presented various scenarios of using our visualization technique to support the analysis of more than seven years of evolution of the Mozilla project.

6.1 Future Work

In the future we plan to explore the following research directions:

Structural information. We want to encapsulate structural information like file size, number of methods, lines of code, *etc.* in the Evolution Radar. The challenge in this is finding a way to encapsulate this data in the Radar layout without losing scalability and readability.

Full integration in BugCrawler. In the current implementation the Evolution Radar is an extension of BugCrawler [6]. By merging the two we will be able to navigate from the structural and evolutionary views provided by BugCrawler to the Evolution Radar.

Sliding time window. We want to use the sliding time window approach, instead of the fixed one, to compute the logical coupling measure.

Bug-related information. We want to apply the same visualization technique using the number of shared bugs as a measure for the dependencies. Our hypothesis is that the greater the number of bugs shared by two entities the stronger their dependency is. We will check this hypothesis by comparing the results obtained using the two measures (*i.e.*, logical coupling and bug sharing).

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the projects "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1), and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997), and the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation Project No. MMI 1976).

7. REFERENCES

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*. IEEE Computer Society Press, Los Alamitos (CA), 2005.

[3] M. Burch, S. Diehl, and P. Weissgerber. Visual data mining in software archives. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 37–46, New York, NY, USA, 2005. ACM Press.

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

[5] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86, New York NY, 2003. ACM Press.

[6] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationships. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages xxx–xxx. IEEE CS Press, Mar. 2006.

[7] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[8] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Los Alamitos CA, 2003. IEEE Computer Society Press.

[9] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, 2004. IEEE Computer Society Press.

[10] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[11] M. Jazayeri, H. Gall, and C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.

[12] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[13] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[14] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[15] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[16] C. Taylor and M. Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Los Alamitos CA, 2002. IEEE Computer Society.

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

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

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

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

An Open Framework for CVS Repository Querying, Analysis and Visualization

Lucian Voinea
Technische Universiteit Eindhoven
Postbus 513, 5600 MB Eindhoven
The Netherlands
Tel. +31402474344
l.voinea@tue.nl

Alexandru Telea
Technische Universiteit Eindhoven
Postbus 513, 5600 MB Eindhoven
The Netherlands
Tel. +31402474344
alext@win.tue.nl

ABSTRACT
We present an open framework for visual mining of CVS software repositories. We address three aspects: data extraction, analysis and visualization. We first discuss the challenges of CVS data extraction and storage, and propose a flexible way to deal with CVS implementation inconsistencies. We next present a new technique to enrich the raw data with information about artifacts showing similar evolution. Finally, we propose a visualization backend and show its applicability on industry-size repositories.

Categories and Subject Descriptors
D.2.7 [**Software engineering**]: Distribution, Maintenance, and Enhancement – *documentation, reengineering*; H.3.3 **[Information Storage and Retrieval**]: Information Search and Retrieval – c*lustering, query formulation*; I.3.8 **[Computer Graphics]**: Applications

General Terms
Management, Measurement, Documentation

Keywords
Evolution visualization, software visualization, CVS repositories

1. INTRODUCTION
Software Configuration Management (SCM) systems are proven instruments for managing large software development projects. SCMs maintain a history of changes in the structure and contents of the managed project. This information is very suitable for empirical studies on software evolution.

Many SCM systems exist on the market, e.g. Subversion, Visual SourceSafe, RCS, CMSynergy, ClearCase and CVS. The Concurrent Versions System (CVS), available via the Open Source community, is a very popular SCM system and has been the preferred choice for SCM support in many Open Source projects in the last decade. Many CVS repositories for long evolution periods, e.g. 5-10 years, are freely available for analysis, so CVS is an interesting option for research on software evolution.

However, CVS is mainly designed for archiving data. CVS offers only a basic querying interface for retrieving a given version of a file or an attribute list with the file state evolution. CVS provides no features to let users get data overviews easily. The user feedback, i.e. state attributes list, is provided only in compiled textual format, which makes it unhandy for quick browsing.

Another challenge of CVS-based software evolution research is the data size and complexity. Raw repository information is too large and provides directly just limited insight in the evolution of a software project. Extra analysis is needed to process these data and extract relevant evolution features.

In this paper we address the challenges of software evolution assessment in CVS repositories. We propose an open framework for CVS data extraction and analysis. We illustrate the capabilities of this framework with a customized implementation. The basic questions we try to answer are:

- How to deal with the large size of CVS data and various limitations of textual feedback?
- How to extract logical coupling information from evolution?
- How to efficiently present evolution data to users to enable correlations across entire projects?

The structure of this paper is as follows. In section 2 we review existing CVS data extraction methods and software evolution analysis techniques. Section 3 presents our flexible interface with CVS repositories. Section 4 describes a new clustering technique for detecting logical coupling of files based on evolution similarity. Section 5 describes a visual back-end for evolution assessment and shows it at work on large repositories. Section 6 summarizes our contribution and outlines open issues for future research.

2. BACKGROUND
The huge potential of the data stored in SCMs for empirical studies on software evolution has been recently acknowledged. The growth in popularity and use of SCM systems, e.g. the open source CVS [5] and Subversion [15], opened new ways for

MSR'06, May 22–23, 2006, Shanghai, China.

project accounting, auditing and understanding. These efforts can be grouped in two directions: data mining and data visualization.

Data mining focuses on processing and extracting relevant information from SCM systems. SCM systems have not been designed to support empirical studies, so they often lack direct access to high-level, aggregated evolution information. Hence, information is distilled from the "raw" stored data by data mining tools, as follows. Fischer *et al*. [7] extend the SCM evolution data with information on file merge points. Gall [9] and German [10] use transaction recovery methods based on fixed time windows. Zimmermann and Weißgerber [21] extended this work with sliding windows and facts mined from commit e-mails. Ball analyzes class-cohesion using a mined probability of classes being modified together [1]. Bieman *et al*. [2] and Gall *et al*. [9] also mine relations between classes based on change similarities. Ying *et al*. [20] and Zimmermann *et al*. [21][21] address relations between finer-grained artifacts, e.g. functions. Lopez-Fernandez *et al*. [13] apply general social network analysis methods on SCM data to assess the similarity and development process of large projects.

Data visualization, the second research direction, takes a different path, focusing on making the large mass of evolution information effectively available to users. Visualization methods make few assumptions on the data – the goal is to let users discover patterns and trends rather than coding these in the mining process. SeeSoft [6], a line-based code visualization tool, uses color to show code snippets matching given modification requests. Augur [8] visually combines project artifact and activity data at a given moment. Xia [19] uses treemap layouts for software structure, colored to show evolution metrics, e.g. time and author of last commit and number of changes. Such tools successfully show the structure of software systems and the change dependencies at given moments. Yet, they don't give insight into code attributes and structure changes made throughout an *entire* project. A first step in this direction, UNIX's `gdiff` and Windows' WinDiff display code differences between two versions of a file by showing line insertions, deletions, and edits computed by the `diff` tool. Still, such tools cannot show the evolution of thousands of files and hundreds of versions. To overcome these shortcomings, Collberg *et al*. [4] depicts the evolution of software structures and mechanisms as a sequence of graphs, for medium-size projects. Lanza [12] depicts the evolution of object-oriented software systems at class level. Wu *et al*. [18] visualize the evolution of entire projects at file level and emphasize the evolution moments. Finally, our own work provided software evolution visualizations at several granularity levels: CVSscan [16] for the line-level evolution of a few source code files and CVSgrab [17] for file-level, project-wide evolution investigations.

Data extraction is a less detailed aspect of software evolution analysis. Many works extract data from CVS repositories, e.g [21], [7], [9], [11], [20], [13], [16], and [17]. Yet, a standard framework for CVS data extraction still lacks. Two main challenges exist here: data retrieval and CVS output parsing. The huge amount of data in CVS repositories is usually available over the Internet. On-the-fly retrieval is not suited for interactive assessment, given the sheer data size. Storing data locally requires long acquisition times, large storage space, and consistency checks. Next, CVS output is ill suited for machine reading. Many CVS systems use ambiguous or nonstandard output formats. Attempts to address these problems exist, but are incomplete. Libraries exist that offer an application interface (API) to CVS, e.g. Java's javacvs or Perl's libcvs. However, javacvs is basically undocumented. Libcvs handles only local repositories. The Eclipse environment offers a CVS client implementation but not an API. The Bonsai project [3] offers several tools to populate a database with evolution data obtained from CVS repositories. These tools are mainly meant as a web data access package and are little documented. The best supported effort for CVS data acquisition is the NetBeans javacvs package [14], a well-documented API with allegedly full CVS client support that parses CVS output into API-level data structures. SoftChange [11] was a first attempt for a coherent environment to support the comparison of Open Source projects, targeting CVS, project mailing lists, and bug report databases. It focuses mainly on data extraction and analysis, aiming to be a generic foundation for building evolution visualization tools.

Overall, several tools exist, each addressing different, though overlapping, facets of software evolution analysis (see Table 1).

Table 1: Tools and methods overview

Tool	Visualization	Query	Analysis
Libcvs	X		
javacvs	X		
Bonsai [3]	X		
Eclipse CVS plugin	X		
NetBeans.javacvs [14]	X		
Release History Database [7]	X	X	
Diff		X	
WinDiff		X	X
eRose [21]	X	X	
QCR [9]		X	
Social Network Analysis [13]		X	
SeeSoft [6]			X
Augur [8]	X		X
Gevol [4]			X
CodeCrawler [12]			X
Evolution Spectograph [18]		X	X
CVSscan [16]		X	X
CVSgrab [17]		X	X
Xia [19]		X	X
SoftChange [11]	X	X	X

We propose a new approach towards an integrated framework for CVS data extraction, analysis and visualization. Our goal is twofold. First, we aim to provide users with a complete software evolution analysis chain. Secondly, we aim at building an experimenting foundation for research at *all* levels, i.e. extraction, analysis, and visualization. Our approach is described next.

3. CVS QUERYING

CVS data extraction is a main problem for research on software evolution. The CVS Internet protocol unfortunately covers only the main CVS function, i.e. file archiving. The CVS navigation commands do not have a machine-readable output. Navigation feedback is given in a compiled text format that is not always easy to decipher. Often, parse tools for this output fail to work on some repositories due to awkward local conventions, e.g. "date format is yyyy-mm-dd and not dd/mm/yyyy" or "file names may contain spaces". This makes uniform access to CVS data difficult. In such cases, one usually searches a parser that copes with the output format at hand and tries to add it to the experimental setup. We propose an approach towards CVS data acquisition that simplifies this process using a data acquisition *mediator* (see Figure 1).

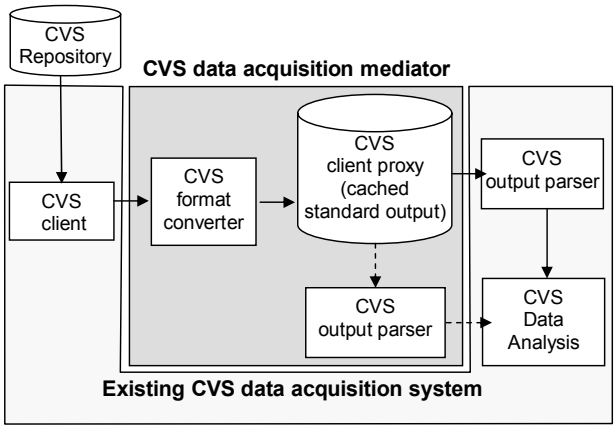


Figure 1: CVS data extractor with output format mediator

The mediator is an easy-to-customize preprocessor between CVS repositories and existing data acquisition tools. When format inconsistencies occur between the CVS output and a parser, we don't need a new CVS data acquisition tool. Instead, we adapt the mediator with a simple rule to transform the new format into the one accepted by the tool. While this doesn't completely remove the problems of inconsistent output formatting, it is a flexible way to solve problems without removing the preferred data acquisition tool. We developed an open source, easy to customize mediator, in a simple to use programming language: python. Secondly, the mediator provides data access to CVS repositories and can also be easily integrated in projects that lack a data acquisition tool. The mediator offers selective access to CVS repositories, i.e. retrieves only information about a desired folder or file, and also caches the retrieved information locally. This design lets one control the trade-off between latency, bandwidth and storage space in the data acquisition step as desired.

4. DATA ANALYSIS

Raw CVS data is too large and low-level to provide insight in the evolution of software projects. Extra analysis is needed to extract relevant evolution aspects. An interesting analysis use-case is to identify artifacts that have similar evolution. Several approaches exist for this [2], [9], [21], [20]. They all use similarity measures based on recovered CVS transactions, i.e. sets of files committed by a user at some moment. The assumption is that related files have a similar evolution pattern, and thus their revisions will often share the same CVS transaction. This information about correlated files is used to predict future changes in the analyzed system, from the perspective of a given artifact.

We propose a more general approach. We argue that not transactions, but pure commit moments, are important for finding similar files. Transaction-based similarity measures fail to correlate files developed by different authors and with different comments attached, but which are still highly coupled. To handle such cases, we propose a similarity measure using the time distance between commit moments. If $S_1 = \{t_i \mid i = 1..N\}$ are the commit moments for a file F_1 and $S_2 = \{t_j \mid j = 1..M\}$ the commit moments for F_2, we define the similarity between F_1 and F_2 as the symmetric sum:

$$\Phi(F_1, F_2) = \sum_{i=1}^{N} \frac{1}{\sqrt{\min\{|t_i - t_j| \mid t_j \in S_2, |t_i - t_j| < k\} + 1}} + \sum_{j=1}^{M} \frac{1}{\sqrt{\min\{|t_j - t_i| \mid t_i \in S_1, |t_j - t_i| < k\} + 1}}$$

where k is a customizable neighborhood factor intended to reduce the influence of completely unrelated events on the similarity measure. The square root is meant to attenuate the influence of the network latency on the CVS transaction. Intuitively, this measure considers, for each commit moment of F_1, the closest commit moment from F_2, weighted by the inverse time distance between the two moments. We next use this measure in an agglomerative clustering algorithm to group files that have a similar evolution, yielding a logical system decomposition following similar evolution patterns. We make the analysis data available for any evolution assessment back-end by storing it in a flat file database.

5. VISUALIZATION

Visualization tries to give insight in these large and complex CVS data by delegating the pattern detection and correlation making to the human visual system. Visualization can also present the results of data analysis in an intuitive, ready-to-use, way. Visualization is a main ingredient of our CVS repository mining framework.

The data acquisition (Sec. 3) and analysis (Sec. 4) steps are generic and can be used with any visualization back-end. We present now a methodology for quick visual assessment of data analysis results and illustrate its applicability with several use cases. For this, we use the CVSgrab tool, detailed in [17]. CVSgrab visualizes project evolution at file level. It depicts each project as a set of horizontal strips representing files along the time axis (Figure 2).

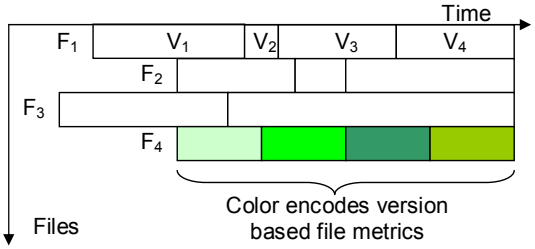


Figure 2: CVSgrab visualization of project evolution

The file layout along the vertical axis is interactively constructed to suit specific analysis needs. Plateau cushions are used to highlight groups of files that have a similar evolution [17]. CVSgrab uses a generic mechanism to map file-level attributes to colors. We next discuss the use of CVSgrab as visualization back-end in our proposed open framework by assessing the evolution of several file metrics on real-life, industry-size CVS repositories. Figure 3 shows the evolution of ArgoUML, an object-oriented design tool with a 6-year evolution of 4452 files developed by 37 authors. To analyze the evolution of ArgoUML using the framework described in this paper, we coupled the CVS data acquisition mediator to the CVSgrab back-end and used the standard CVS client to access the ArgoUML repository over the Internet. Data acquisition took 31 minutes over a T1 Internet connection: 8 minutes for the initial setup (i.e. one-time retrieval of the last version of 56MB) and 23 minutes to retrieve the evolution data to be visualized (29MB).

In Figure 3, a 12-snapshot matrix shows ArgoUML's evolution. Each column shows the evolution of one metric:

- Column 1 shows the development team evolution. Each file version color shows the ID of the user who committed it.
- Column 2 shows the size evolution of contributions as number of lines. Files that are first committed are colored gray. Blue shows file size increase, red is decrease, and yellow is a commit that affects several lines but does not modify the file size. While hue encodes the type of change in size, brightness encodes the change size: lighter colors denote smaller changes, darker colors denote more modifications.
- Column 3 shows the file type: red for java source files, green for images, and yellow for HTML files.
- Column 4 highlights versions that contain given strings in their associated commit comment: green for versions that contain the word "fix", blue for versions that contain the word "error".

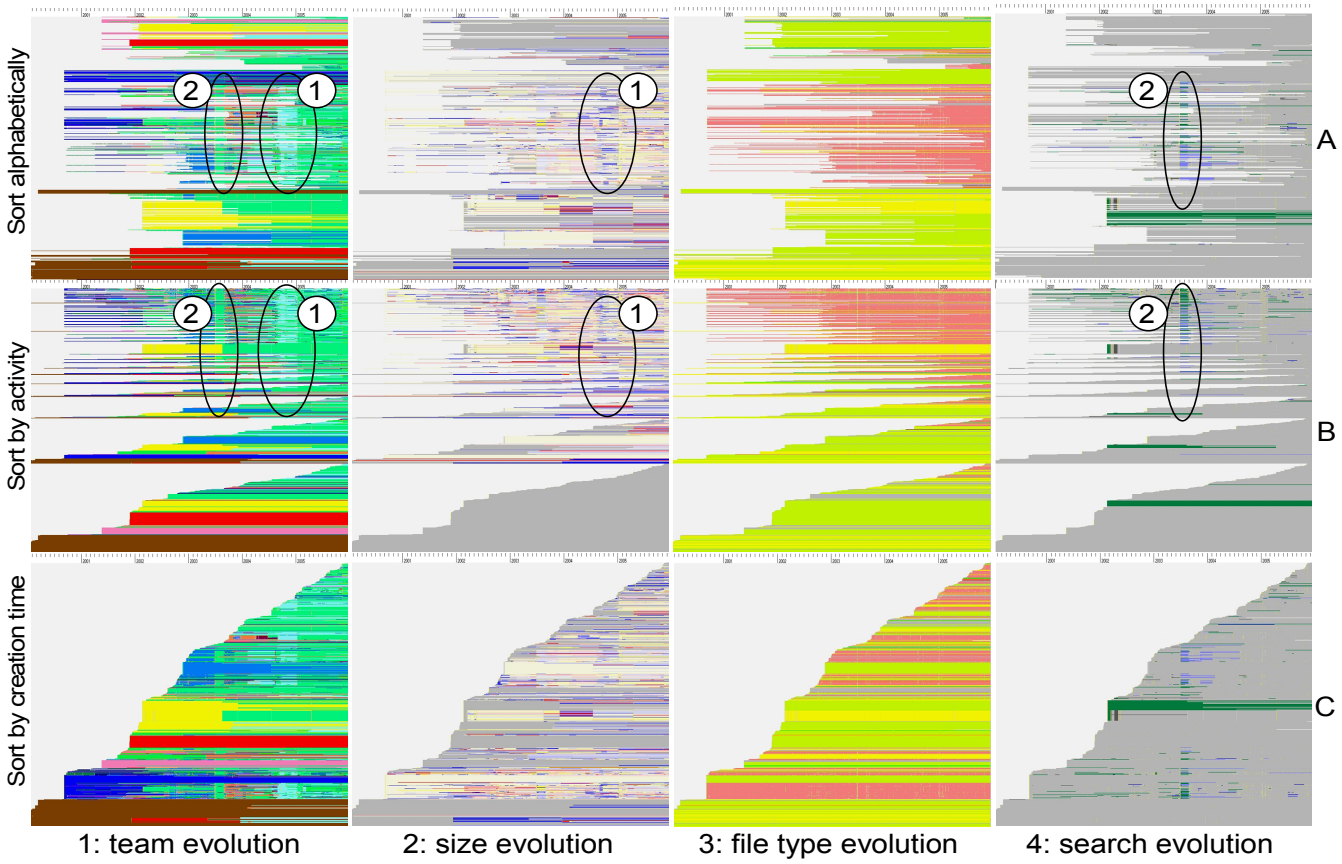


Figure 3: ArgoUML metrics evolution visualization with CVSgrab

Each row in Figure 3 uses another layout offered by CVSgrab. In row A, files are sorted alphabetically on their full path, and thus show the folder structure. In row B, files are sorted from top to bottom in decreasing order of number of versions, i.e. file activity. Files that have the same number of versions are further sorted in decreasing order of creation time. In row C, files are sorted in decreasing order of creation time. Files created in the beginning of the project are at the bottom, while 'young' files are at the top.

By assessing the project evolution shown in Figure 3, one can discover several interesting aspects of the process and organization of ArgoUML. Cell C3 shows that the project started with a documentation base (i.e. green and yellow) that probably contained the system specification. This was contributed by one user (*jrobbins* = brown in C1) and remained unchanged for the entire project duration except for a large addition (dark blue in C2) done by another user two years later (*dennyd* = red in C1). The added code concerned seemingly an underspecified issue as it was extend again two years later (dark blue in C2) by another

author (*mvw* = cyan in C1). The real implementation first appeared 6 months after the specification was committed (java source files = red in C3) and was contributed by one author (*1sturm* = blue in C1). Two years from the project start, another big documentation chunk was added (yellow and green in C3) by one user (*jeremybennett* = yellow in C1). Although both the specification, implementation, and documentation parts appear to be the work of one author each, it is intriguing the fact they were all committed at one time (i.e. not incrementally), by one author, and contained many files, i.e. approx. 400. It is thus possible that these represent the work of more people, which was first checked in by one single person. For the rest of the project, one user has a significant contribution (*linus* = green in C1), with one exception in the fifth project year (*mvw* = cyan in C1). The large oval (1) in A1 shows that *mvw* (cyan in A1) made a significant contribution (dark blue, large oval (1) in A2) to the implementation (red in A3). The same pattern can be recognized following the large ovals (1) in B1 and B2. A3 shows that the project has a very clean organization. The major color groups correspond to the folders *documentation* (green at the top), *src_new* (red at the middle) and *www* (yellow and green at the bottom). From B3, one can see that most activity during the project was related, as expected, to changes in the implementation files (red at the top) followed by changes in the documentation (yellow in the middle) and in the documentation images (green at the bottom). B2 shows that almost one-third of the files added during the project did not change during all six years (since they are grey). Most such files contain documentation (i.e. yellow and green in B3). To this group belongs also the largest part of the previously identified system specification (i.e. brown in B1, by correlation with C1 and C3). Another interesting aspect is shown in the small ovals (2) in A1 and A4. It seems that in the fourth project year *linus* made a significant contribution, not in terms of size (i.e. no significant size change pattern detected in A2) but in terms of code cleaning. Many implementation files (red in A3) containing the words "fix" and "error" in their revision comment have been committed by *linus* to the repository. The same pattern can be seen in row B. The large green (i.e. "fix") horizontal pattern that can be seen in column 4 corresponds to an initial checkout of documentation files. It suggests that previous work has been done in that area without being committed. Figure 3 shows also that almost no significant decrease took place in the project size. One exception, highlighted in C2, shows a size drop for documentation files (yellow in C3) that occurred at the end of the fourth project year.

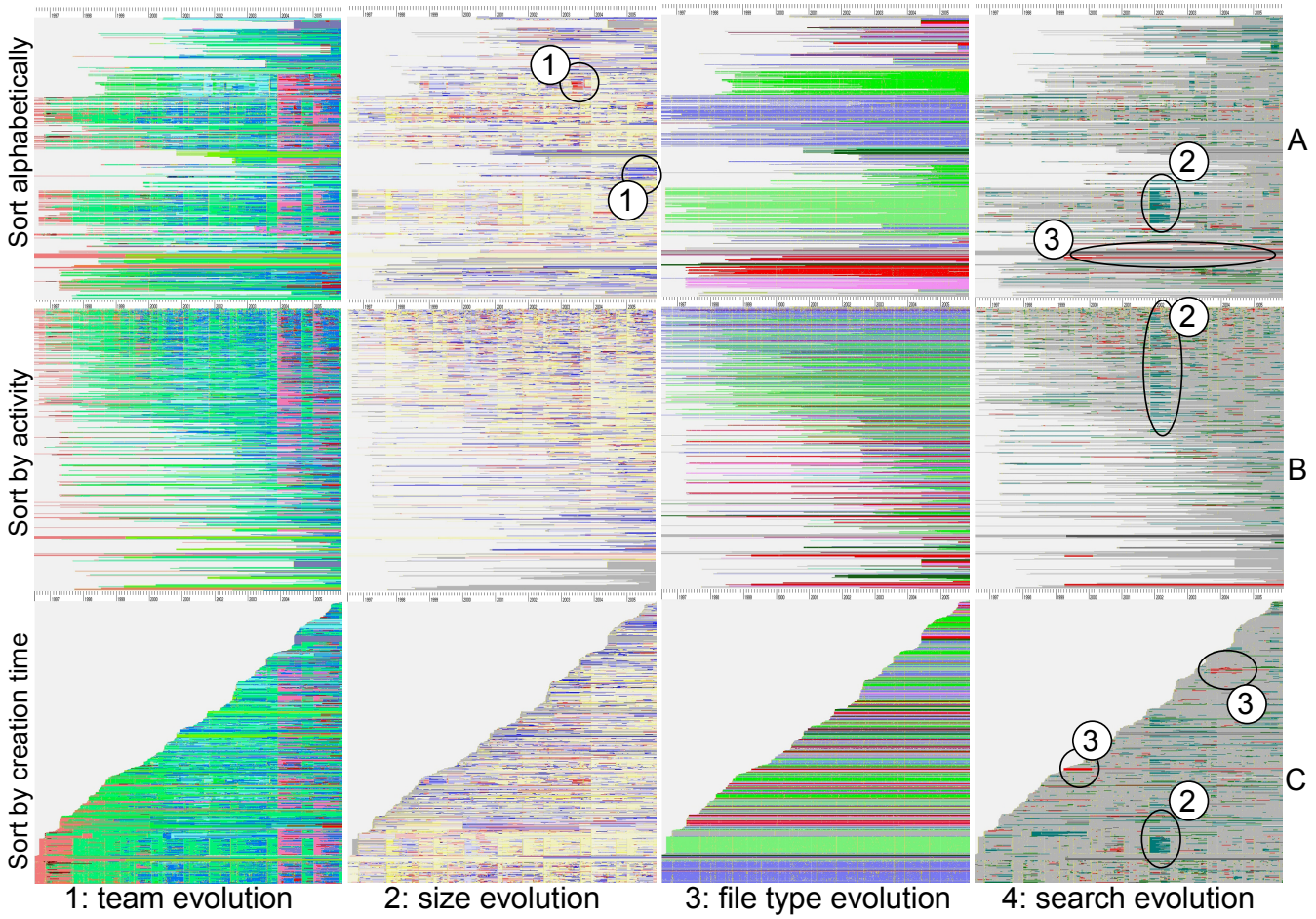


Figure 4: PostgreSQL metrics evolution visualization with CVSgrab

Figure 4 gives another example of CVS evolution visualization done using our proposed framework. It shows the evolution of PostgreSQL, an object-relational database management system project with a history of 10 years, 2829 files, and 27 authors. We used the same framework setup as in the previous example. The data acquisition step took 28 minutes: 7 minutes for the initial setup (i.e. one time retrieval of the last project version = 56MB) and 21 minutes for retrieving the evolution information to be

visualized (29MB). The evolution retrieving time was in this case smaller than in the first example, even if more data was retrieved. This is explained by the connection overhead. When retrieving evolution data, the connection has to be established for each file. In this case the number of files was less than in the first example, which significantly improved the overall connection latency.

Figure 4 shows 12-snapshot matrix illustrating the evolution of PostgreSQL, structured similarly to Figure 3. Columns show the development team (1), size evolution (2), file type (3) and string search (4) encoded by colors, just as in example 1, except for file type and string search. In column 3, C source files are blue, light C headers are light green, SGML documentation files are normal green, SQL files are pink, and test support files are red. In column 4, green shows versions that contain the word "fix" in their associated commit comment, and red versions that contain the word "bug". As in the first example, the matrix rows use different sortings for arranging files on the vertical axis: alphabetical order (A), number of revisions (B) and creation time (C).

Assessing the evolution information depicted in Figure 4 one can compare the evolution of PostgreSQL with the one of ArgoUML presented in the first example, as follows. Cell C3 shows that the project started with a source code base (i.e. blue at the bottom) and not with a specification, as for ArgoUML. Even the header files containing interfaces were not fixed until a couple of months later (light green). As for ArgoUML, the initial contribution to the repository (C source and headers) was performed by one person (*scrappy* = red in C1) and incorporated many files (approx. 400). This suggests that previous developments existed that were not recorded in CVS. The rest of the evolution appears to be mainly the contribution of a few authors: *momjian* (light green), *tgl* (dark blue), *pgsql* (magenta), *petere* (cyan), *thomas* (yellow-greenish). The contributions of *momjian* and *tgl* are interleaved at periods of around 6-8 months (column 1) and address the most active parts of the system (B1). These parts correspond to the implementation files (i.e. C source code and headers), by correlation via A1 and A3. These parts are also targeted by *pgsql* in the last two project years. A detailed look at B1 and B2 reveals the contribution patterns of *momjian* and *tgl*. The versions committed by *momjian* do not usually bring changes in files sizes (i.e. they are yellow in B2) and are relatively done at large intervals. In contrast to this, the contributions of *tgl* are done at smaller intervals and cause often changes in the file size. Moreover, the contributions of *momjian* "interrupt" abruptly the ones of *tgl* but not conversely. This suggests the real work might be done by *tgl* while *momjian* has more the role of a code standard manager. A more in-depth investigation of the evolution using the details-on-demand mechanism of CVSgrab showed that the modifications done by *momjian* addressed mainly changes in indentation and copyright texts. A similar pattern holds for *pqsql*. Finally, p*etere* and *thomas* appeared to have mainly contributed to the system documentation (by correlating A1 and A3). As for ArgoUML, PostgreSQL seems to have a clean organization (A3): Source, header, documentation, and test files are well separated. Most of the activity takes place in the implementation files. Not only C files are modified but also headers and documentation files, which could suggest frequent architectural changes. No significant size modifications are registered throughout the project. The only exceptions, (1) highlighted in A2, address the documentation part of the project. Finally, column 4 in Figure 4 shows the distribution of the words "fix" and "bug" along the project evolution. The green patterns (2) highlighted in the image correspond to versions containing the word "fix". By correlating C3, C4 and C1, it seems that these patterns match header files in versions committed by *momjian*. Hence, it is possible they do not address important changes for the system functionality. Indeed, a more detailed analysis revealed that the word "fix" refers actually to a version of an indentation program used to format the text and not to the system code itself! Other occurrences of the word "fix" are evenly distributed largely over the evolution of C source files (A4). The red patterns (3) highlighted in A4 and C4 correspond to versions containing the word "bug". They correspond to test files and appear towards the file creation moment. This, together with the fact that test files are created relatively early in the project, suggests an active test policy. The rest of the occurrences of the word "bug" are evenly distributed, mostly along the evolution of C source files.

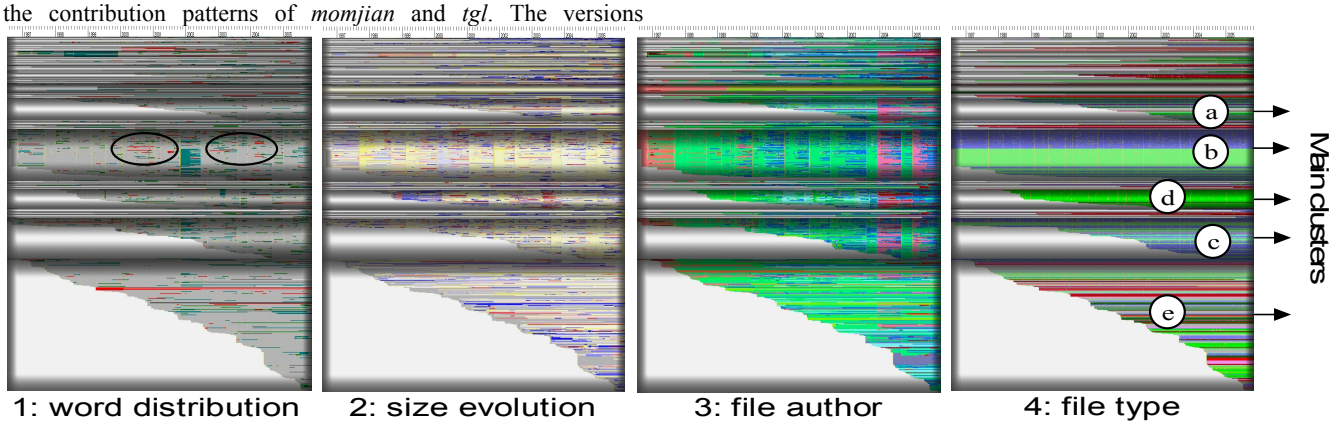


Figure 5: PostgreSQL evolution clusters visualization with CVSgrab

Figure 5 visualizes PostgreSQL evolution enriched with data analysis about clusters of files with common evolution. Four CVSgrab snapshots are presented. Clusters are encoded using plateau cushions. In each cluster, files are sorted in decreasing order of their creation time, from top to bottom. Color shows different file metrics: word distribution (1), size evolution (2), file author (3) and file type (4), as in Figure 4. There are mainly five important evolution groups. In column 4, one can see three main groups: source files (a,b,c), documentation (d), and test scenario files (e). We can easily see that source files introduced in the beginning of the project have a similar evolution (b). Hence, they may refer to a part of the system that has a high logical coupling

and can be seen as a building block. The same holds for the other two clusters containing source code (a, c). All building blocks share the same developer network (3) and size evolution patterns (2). The block corresponding to the early introduced source code (b) has, however, a higher density of versions with comments containing the word "bug" (highlighted in 1). Hence, this block may contain a problematic implementation. Documentation forms a separate cluster (d), leading to the conclusion that it mainly targets the functionality of the system and not its detailed design, as it doesn't change in sync with the headers. Finally, the large cluster at the bottom of the images (e) corresponds to a miscellaneous collection of files including test scenarios. This cluster may thus refer to files intended to support the development process, rather than implementing real functionality.

Changing the cluster granularity level, one can further split the clusters presented above for a finer analysis of the system. This can be useful not only for performing a logical decomposition, but also for predicting future changes with different levels of confidence.

6. CONCLUSIONS

In this paper we propose a new framework for visual data mining of CVS software repositories. Our goal is twofold. On the one hand we aim to provide the research community with a base for experimentation of new techniques in data acquisition, analysis and visualization. On the other hand, we want to increase the framework acceptance by making it immediately available to the end users for CVS repository mining.

To achieve the first goal we propose a mediator module for CVS data acquisition that can easily integrate with current data extraction systems. The role of this module is to facilitate the resolution of CVS format incompatibility problems without requiring the modification / replacement of the data acquisition module. Secondly, we propose a new approach for quick visualization of data analysis results using the generic metric visualization mechanism of CVSgrab [17].

To make the framework immediately available to end users, we integrate the CVS mediator with a reference implementation of a data extraction tool. Additionally, we propose a new technique for identifying clusters of files with similar evolution. This could help users both to perform a logical decomposition of the system, and to predict future changes in the system from the perspective of select files. We integrate this technique as a data analysis module in the proposed framework, and we use CVSgrab [17] as visualization backend. Finally we illustrate the functionality of the integrated framework by visually mining the evolution of two industry-size Open Source projects: ArgoUML and PostgreSQL. The two cases demonstrate the framework has affordable time, bandwidth, and storage requirements for data acquisition. Additionally, it enables users to easily make complex evolution assessments by correlating evolution of multiple file metrics.

As a future direction of research we would like to improve the similarity measure of the evolution clustering mechanism by using additional attributes, e.g. file type, author. The challenge in this direction is to find the best similarity description that matches a given user requirement. Additionally, we would like to extend the framework with other generic visualization mechanisms, for easy assessment of data analysis techniques.

7. REFERENCES

[1] Ball, T., Kim, J.-M., Porter, A.A., and Siy, H.P. If your version control system could talk. *ICSE'97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997. http://research.microsoft.com/~tball/papers/icse97-decay.pdf

[2] Bieman, J. M., Andrews, A. A., and Yang, H. J. Understanding change-proneness in OO software through visualization. *Proc. Intl. Workshop on Program Comprehension*, IEEE Press, 2003, pp. 44–53

[3] Bonsai online: http://www.mozilla.org/projects/bonsai/

[4] Collberg, C., Kobourov, S., Nagra, J., Pitts, J., and Wampler, K. A System for Graph-Based Visualization of the Evolution of Software. *Proc. ACM SoftVis'03*, ACM Press, 2003, pp. 77–86

[5] CVS online: http://www.nongnu.org/cvs/

[6] Eick, S.G., Steffen, J.L., and Sumner, E.E. Seesoft - A Tool For Visualizing Line Oriented Software Statistics. *IEEE Trans. on Software Engineering*, 18:11, IEEE Press, 1992, pp. 957– 968

[7] Fischer, M., Pinzger, M., and Gall, H. Populating a Release History Database from version control and bug tracking systems. *Proc. Intl. Conf. on Software Maintenance*, IEEE Press, 2003, pp. 23–32

[8] Froehlich, J., and Dourish, P., Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. *Proc. ICSE'04*, IEEE Press, 2004, pp.387–396

[9] Gall, H., Jazayeri, M., and Krajewski, J. CVS release history data for detecting logical couplings. *Proc. IWPSE'03*, IEEE Press, 2003, pp. 13–23

[10] German, D., and Mockus, A. Automating the measurement of open source projects. *ICSE '03 Workshop on Open Source Software Engineering*, *Automating the Measurement of Open Source Projects*, http://www.research.avayalabs.com/user/audris/papers/oose03.pdf

[11] German, D., Hindle, A., and Jordan, N. Visualizing the evolution of software using softchange. In *Proc. Intl .Conference on Software Engineering and Knowledge Engineering* (SEKE'04), pp. 336–341

[12] Lanza, M. The evolution matix: Recovering software evolution using software visualization techniques. In *Proc. Intl. Workshop on Principles of Software Evolution*, ACM Press, 2001, pp. 37–42

[13] Lopez-Fernandez, L., Robles, G., and Gonzalez-Barahona, J.M. Applying Social Network Analysis to the Information in CVS Repositories. *Intl. Workshop on Mining Software Repositories (MSR)*, 2004, http://opensource.mit.edu/papers/llopez-sna-short.pdf

[14] NetBeans.javacvs online: http://javacvs.netbeans.org/

[15] Subversion online: http://subversion.tigris.org/

[16] Voinea, L., Telea, A., and van Wijk, J.J. CVSscan: Visualization of code evolution. *Proc. ACM SoftVis*, ACM Press, 2005, pp. 47 – 56

[17] Voinea, L., and Telea, A. CVSgrab: Mining the History of Large Software Projects. *Proc. EuroVis'06*, IEEE Press, 2006.

[18] Wu, K., Spitzer, C.W., Hassan, A.E., and Holt, R.C. Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution. In *Proc. Intl. Workshop on Principles of Software Evolution* (IWPSE'04), IEEE Press, 2004, pp. 57-66

[19] Wu, X. *Visualization of version control information*. Master's thesis, University of Victoria, 2003.

[20] Ying, A.T.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C., Predicting Source Code Changes by Mining Revision History. *IEEE Trans. on Software Engineering*, 30:9, IEEE Press, 2004, pp. 574-586

[21] Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A., Mining version histories to guide software changes. *Proc. Intl. Conference on Software Engineering* (ICSE), IEEE Press, 2004, pp. 429–445

[22] Zimmermann, T., Weißgerber, P., Preprocessing CVS Data for Fine-grained Analysis. *Intl. Workshop on Mining Software Repositories (MSR)*, May 2004, http://www.st.cs.uni-sb.de/papers/msr2004/msr2004.pdf

Micro Pattern Evolution

Sunghun Kim
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
hunkim@cs.ucsc.edu

Kai Pan
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
pankai@cs.ucsc.edu

E. James Whitehead, Jr.
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA, USA
ejw@cs.ucsc.edu

ABSTRACT

When analyzing the evolution history of a software project, we wish to develop results that generalize across projects. One approach is to analyze design patterns, permitting characteristics of the evolution to be associated with patterns, instead of source code. Traditional design patterns are generally not amenable to reliable automatic extraction from source code, yet automation is crucial for scalable evolution analysis. Instead, we analyze "micro pattern" evolution; patterns whose abstraction level is closer to source code, and designed to be automatically extractable from Java source code or bytecode. We perform micro-pattern evolution analysis on three open source projects, ArgoUML, Columba, and jEdit to identify micro pattern frequencies, common kinds of pattern evolution, and bug-prone patterns. In all analyzed projects, we found that the micro patterns of Java classes do not change often. Common bug-prone pattern evolution kinds are 'Pool → Pool', 'Implementor → NONE', and 'Sampler → Sampler'. Among all pattern evolution kinds, 'Box', 'CompoundBox', 'Pool', 'CommonState', and 'Outline' micro patterns have high bug rates, but they have low frequencies and a small number of changes. The pattern evolution kinds that are bug-prone are somewhat similar across projects. The bug-prone pattern evolution kinds of two different periods of the same project are almost identical.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms

Algorithms, Measurement, Experimentation

1. INTRODUCTION

Software evolution research examines the development history of a software project to learn facts about the software, and better understand its qualities. After examining the history of many different software projects, ideally we would like to be able to make claims like, if we observe evolution pattern *X*, then the consequences for one or more software qualities are *Y* and *Z*.

MSR '06, May 22-23, 2006, Shanghai, China.

Most software repository mining research examines software by subdividing it into parts using physical distinctions, such as modules, directories, files, and methods. Researchers examine the evolution of these physical elements, and then correlate various software properties with traits of the observed evolution. For example, researchers have examined revision histories to determine correlations between changes and bugs [13]. Though there has been much success in correlating software properties with the evolution of physical elements within a project, the ability to apply these results to other projects has been limited. This is due to the use of the software's existing physical distinctions, which limits the applicability of results to just a single project. Knowing something about the evolution of the methods in a specific Java class does not typically provide any insight into other classes, since different classes have different source code.

To make more generalizable observations requires some means for abstracting away from the physical elements into abstract categories. These categories need to be concrete enough to capture important aspects of the behavior of the software, yet sufficiently general that one can observe the same abstract categories across multiple projects. The classic software design patterns [6] fit this description, and suggest the possibility that we can deeply understand the evolutionary behavior of specific design patterns. To perform such analysis in a scalable way, we need an automated mechanism for extracting software design patterns from source code. Unfortunately, to date there is no accurate mechanism for identifying design patterns in code, with existing approaches suffering from large amounts of false positives or false negatives.

Recent work by Gil and Maman has introduced the concept of micro patterns [7], which are "Java class-level traceable patterns." These are more fine-grained design patterns than the classic patterns, and have been designed to always be automatically extractable from source code (or bytecode). Micro patterns express more fine-grained design idioms than classic patterns. For our purposes, what is important is that we now have a reliable, automatic way to extract a set of general design abstractions from Java projects. This now allows us to explore whether evolution characteristics can be correlated with the abstractions inherent in these micro patterns, and make generalizable conclusions about specific evolution patterns.

In this paper we analyze the micro pattern evolution of three open source projects, ArgoUML, Columba, and JEdit, shown in Table 1. Our goal in doing so is to examine whether there are any correlations between the evolution of micro patterns and the likelihood of having bugs. Ideally we wish to identify micro pattern evolution kinds that are consistently fault prone across projects, and hence allow us to make general conclusions about this kind of evolution that have broad applicability.

Table 1. Analyzed projects, ArgoUML, Columba, and jEdit. # of revisions is the number of revisions we analyzed. # of class changes indicates the number of corresponding source code (Java) changes. # of bug changes indicates the number of changes that introduce bugs identified by mining change logs and SCM history [13]. % of bug rate is the rate of bug-introducing changes over all changes.

Project	Software type	Period	# of revision	# of class changes	# of bug class changes	% of bug rate
ArgoUML	UML design tool	01/2002 ~ 03/2003	1,262	4,179	1,245	29.8
Columba	Email Client	11/2002 ~ 01/2006	1,652	11,138	1,604	14.4
jEdit	Editor	09/2001 ~ 01/2006	1,449	5,526	2,456	44.5

After examining the micro pattern evolution history of the three open source projects, we found that micro patterns do not typically change when a class file changes. For example, the most common pattern evolution kinds are 'Limited Self → Limited Self', 'Implementor → Implementor', and 'Sink → Sink' (these micro patterns are briefly described in Section 2). In all these cases the micro pattern is the same before and after the class change. Only 4-6% of class file changes cause micro pattern changes, examples being 'Implementor → NONE' and 'Stateless → RestrictedCreation'.

For each project we identified the micro pattern kinds that were most bug-prone. We additionally found the most bug-prone pattern evolution kinds of the three projects, and found that they are somewhat similar. Furthermore, we observed that the bug-prone evolution kinds for two different periods of the same project are almost identical. For example, micro pattern evolution kinds such as 'Pool → Pool', 'Implementor → NONE', and 'Sampler → Sampler' are bug-prone in jEdit. We found that 'Box', 'CompoundBox', 'Pool', 'CommonState', and 'Outline' micro patterns have high bug rates, but they have low frequencies and a small number of changes. In contrast, 'Overrider' and 'Sink' micro patterns have relatively lower bug rates.

We anticipate that these findings can be used by software quality engineers to identify areas of a software project that are more bug-prone, and apply more testing and verification resources to those areas. We could also make software developers aware that they are working on a bug-prone pattern, or kind of pattern transition, and thereby encourage more defensive coding and more extensive unit testing.

In the remainder of the paper, we explain micro patterns (Section 2) and describe our experimental setup (Section 3). Following are results from our experiments (Section 4), along with discussion of the results (Section 5). Rounding off the paper, we end with related work (Section 6) and conclusions (Section 7).

2. JAVA MICRO PATTERNS

Micro patterns capture idioms of Java programming languages such as the use of inheritance, immutability, data wrapping, data management, and modularity [7]. Micro patterns include Box, Compound Box, Sampler, Canopy, Immutable, Implementor, Pseudo Class, Pool, Restricted Creation, Overrider, Sink, Stateless, Common State, Outline, Function Pointer, Function Object, Joiner, Designator, Record, Taxonomy, PureType, Augmented Type, Extender, Data Manager, Trait, Cobol Like, State Machine Recursive, and Limited Self [7]. While the reader is strongly encouraged to examine [7] for a detailed description, we describe a few micro patterns here to provide a flavor of these patterns:

Pool: A class has only final static fields and no methods.

Box: A class has exactly one instance field, which can be modified by methods in the class.

Sampler: A class that has at least one public constructor and at least one static field whose type is the same as that of the class.

Limited Self: Suppose class '*foo*' is a subclass of class *'bar'*. If *'foo'* does not introduce any new fields, and all self method calls in *'foo'* are calls to methods in *'bar'*, then *'foo'* is a Limited Self pattern class.

Recursive: A class that has at least one field whose type is the same as that of the class. For example, java.util.LinkedList is a recursive pattern class.

Sink: a class whose declared methods do not call instance methods or static methods.

Implementor: a non-abstract class such that all of its public classes are implementation of its super abstract class.

We use micro patterns for our pattern change analysis for three reasons: (1) they are traceable, (2) they are close to the source code, (3) and they capture non-trivial design idioms of the Java language.

3. EXPERIMENT SETUP

In this section, we describe the data used in our study and explain how it was extracted. We use the Kenyon [3] infrastructure to automatically extract project revisions and class changes from the SCM repositories for ArgoUML, Columba, and jEdit. Bug-introducing changes are identified by mining change logs and project history data using techniques described in [13]. Micro patterns are extracted using a pattern extraction tool developed by Gil and Maman [7], after compiling each revision.

3.1 Micro Pattern Extraction

We extract software histories including all revisions and all files from SCM systems such as CVS [2] using the Kenyon infrastructure [3]. After checking out each project revisions, we compile the revision and generate a jar file. We feed the jar file into the micro pattern extraction tool [7]. The tool automatically reads all class files in the jar file and extracts the pattern(s) matched by each class file. We persistently store these extracted micro patterns for each Java class file for all revisions of all three projects.

3.2 Pattern Changes and Bug Changes

Now we have the micro patterns for all Java class files (corresponding Java source files) of each revision. Using the standard *diff* tool, we can easily identify Java class file changes. To determine bug-introducing changes, we mine SCM change logs and project history data [13]. We then observe the micro pattern changes in each Java class file and compute bug introduction rates for these changes.

For example, consider the change history for the file '*foo.java*' (*foo.class*) as shown in Figure 1. The change log at revision 6 (Rev 6) states "Fixed issue #355", which indicates that it is a fix change. It means the file at revision 5 has one or more problematic lines, which are fixed in revision 6 by changing the problematic lines. When were the problematic lines added in the first place? SCM systems such as CVS [2] and Subversion [1] provide an annotation feature that shows information about when each line of a file was modified, and by whom. Using SCM annotation, we can find out when the problematic lines were initially added. Suppose the problematic lines were added in revision 3. This means the file at revision 2 does not have the problematic lines, so they were added in the change between revision 2 and 3. This change introduced a bug into the software, and hence we call it a bug-introducing change.

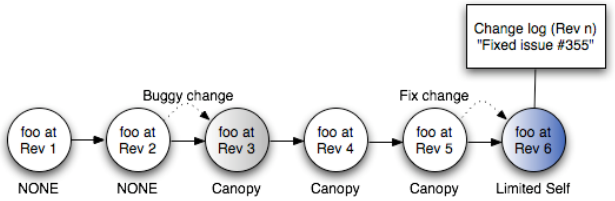


Figure 1. Example of pattern evolution kinds and a bug-introducing change.

The micro patterns for each revision are shown in Figure 1. As an example, the pattern evolution kind for '*foo.class*' between revisions 1 and 2 is 'NONE → NONE'. In Figure 1, we see the following micro pattern evolution kinds: '*NONE → NONE*' (1 time), '*NONE → Canopy*' (1 time), '*Canopy → Canopy*' (2 times), and '*Canopy → Limited Self*' (1 time). We count the number of bug-introducing changes and compute the bug introduction rate for each micro pattern evolution kind. For example, the bug-introducing change count of the '*NONE → Canopy*' kind is 1, and occurs 1 time, so it has a 100% bug introduction rate. General categories of pattern evolution kinds are described in Table 2.

Table 2. Categories of pattern evolution kinds

Category		Description
Pattern unchanged		Pattern remains the same after a class change. E.g. Canopy → Canopy
Pattern changes	Change to other pattern	Pattern changes to other patterns. E.g. Stateless → RestrictedCreation
	Losing pattern	Pattern changes to NONE. E.g. LimitedSelf → NONE
	Adding pattern	Pattern changes from NONE. E.g. NONE → Stateless

When we compute the bug introduction rates of micro pattern evolution kinds, we filter out the total count if it is less than 10 (outliers). If a micro pattern evolution kind occurs less than 10 times, we believe it is hard to make general conclusions about its bug introduction rate, and it is possible that a small number of bugs can affect the bug introduction rate substantially.

4. RESULTS

We first present micro pattern frequencies of a project snapshot (the latest revision). We next show the list of micro pattern evolution kinds, their counts, and ratios. The bug-prone micro pattern evolution kinds are shown using contour graphs. We compare common bug-prone evolution kinds of three projects and two periods of the same project. Finally, we compare frequencies, the number of changes, and bug rates of each micro pattern.

Table 3. Java micro pattern frequencies of analyzed projects. The * marked patterns do not exist or are rare in the analyzed projects; we exclude them in further analysis.

Micro Patterns	ArgoUML (%)	Columba (%)	jEdit (%)
Box	1	3	2
Compound Box	1	2	6
Sampler	1	2	1
Canopy	8	10	22
Immutable	3	5	10
Implementor	28	31	32
*Pseudo Class	0	0	0
Pool	1	3	1
Restricted Creation	2	2	1
Overrider	8	7	22
Sink	5	10	10
Stateless	8	9	5
Common State	1	1	3
Outline	1	0	0
Function Pointer	1	2	1
Function Object	5	7	19
*Joiner	0	0	0
*Designator	0	0	0
Record	0	0	1
Taxonomy	1	2	2
PureType	4	9	4
*Augmented Type	0	0	0
Extender	3	11	5
Data Manager	1	3	1
*Trait	0	0	0
Cobol Like	0	1	0
State Machine	1	2	1
Recursive	0	0	2
Limited Self	16	20	12
Coverage	55	79	81

4.1 Pattern Frequencies

We compute micro pattern frequencies of a project snapshot (the latest revision), with results shown in Table 3. 'Canopy', 'Implementor', 'Overrider', 'Function Object', and 'Limited Self' are the most prevalent micro patterns. 81% of classes have one or more micro patterns in jEdit, 79% for Columba and 55% for ArgoUML. The remaining classes do not match any micro pattern (NONE). Some micro patterns, such as 'Joiner' or 'Pseudo Class,' do not exist in the latest revision.

The micro pattern distributions of three projects are quite similar. For example, the Pearson's correlation coefficient [5] of the micro pattern frequencies of ArgoUML and jEdit is 0.96. Even though these two projects have different physical features in their source code, they have similar pattern frequencies, suggesting that any correlations between patterns, or pattern evolution kinds found in these two projects would have applicability to both projects, and perhaps others as well.

4.2 Pattern Evolution Kinds

We count all micro pattern evolution kinds of each Java class file change across the project histories. Table 4 shows the top 20 micro pattern evolution kinds, their counts, and relative frequency (percentage of all observed pattern evolutions) of each pattern change. The most common micro pattern evolution kind is 'NONE → NONE'. Other common micro pattern evolution kinds are 'LimitedSelf → LimitedSelf', 'Implementor → Implementor', 'Overrider → Overrider', and 'Extender → Extender'. The common micro pattern evolution kinds are similar for the three projects.

Table 4. Top 20 most common pattern evolution kinds of the analyzed projects.

Rank	ArgoUML Pattern evolution kind	ArgoUML change # (change %)	Columba Pattern evolution kind	Columba change # (change %)	jEdit Pattern evolution kind	jEdit change # (change %)
1	NONE → NONE	1830 (33%)	NONE → NONE	4245 (31%)	NONE → NONE	1738 (24%)
2	LimitedSelf → LimitedSelf	931 (17%)	LimitedSelf → LimitedSelf	1684 (12%)	LimitedSelf → LimitedSelf	803 (11%)
3	RestrictedCreation → RestrictedCreation	490 (8.8%)	Implementor → Implementor	1589 (12%)	Overrider → Overrider	751 (10%)
4	Implementor → Implementor	375 (6.8%)	Extender → Extender	1294 (9.5%)	CommonState → CommonState	582 (7.9%)
5	Overrider → Overrider	342 (6.2%)	Overrider → Overrider	749 (5.5%)	Implementor → Implementor	575 (7.8%)
6	Extender → Extender	262 (4.7%)	Stateless → Stateless	578 (4.2%)	Canopy → Canopy	452 (6.2%)
7	Sink → Sink	203 (3.7%)	Sink → Sink	538 (4%)	Recursive → Recursive	317 (4.3%)
8	Stateless → Stateless	189 (3.4%)	CommonState → CommonState	237 (1.7%)	Extender → Extender	286 (4%)
9	Sampler → Sampler	142 (2.6%)	Immutable → Immutable	223 (1.6%)	Sampler → Sampler	274 (3.7%)
10	Common State → Common State	91 (1.6%)	Box → Box	201 (1.5%)	Immutable → Immutable	223 (3%)
11	Immutable → Immutable	77 (1.4%)	PureType → PureType	178 (1.3%)	CompoundBox → CompoundBox	216 (2.9%)
12	Compound Box → Compound Box	70 (1.3%)	Taxonomy → Taxonomy	163 (1.2%)	FunctionObject → FunctionObject	183 (2.5%)
13	Implementor → NONE	70 (1.3%)	DataManager → DataManager	163 (1.2%)	Sink → Sink	170 (2.3%)
14	NONE → Stateless	50 (0.9%)	CompoundBox → CompoundBox	161 (1.2%)	Pool → Pool	140 (1.9%)
15	Canopy → Canopy	45 (0.8%)	Canopy → Canopy	145 (1.1%)	Stateless → Stateless	112 (1.5%)
16	Outline → Outline	42 (0.8%)	Outline → Outline	143 (1%)	PureType → PureType	63 (0.9%)
17	Box → Box	30 (0.5%)	RestrictedCreation → RestrictedCreation	127 (0.9%)	Box → Box	39 (0.5%)
18	LimitedSelf → NONE	27 (0.5%)	FunctionPointer → FunctionPointer	114 (0.8%)	DataManager → DataManager	37 (0.5%)
19	Pool → Pool	25 (0.5%)	Pool → Pool	97(0.7%)	Outline → Outline	24 (0.3%)
20	NONE → Implementor	24 (0.4%)	FunctionObject → FunctionObject	82 (0.6%)	Taxonomy → Taxonomy	17 (0.2%)

Also note that the patterns in the top pattern evolution kinds are not the same as the most frequent patterns shown in Table 3. For example, the most common pattern in jEdit is 'Implementor', but the most common pattern evolution kind is 'LimitedSelf → Limited Self' (excluding 'NONE → NONE'). The fourth ranked pattern evolution kind, 'CommonState → CommonState', is a relatively rare micro pattern in jEdit (only 3%).

Overall, micro patterns in Java class files do not frequently transition to new micro patterns. If a Java class file exhibits characteristics of a given micro pattern, the class file tends to stick to the original micro pattern as the class file changes. Table 5 shows the counts and percentages of pattern evolutions that change patterns, and those that do not. Only 4 to 6% of Java class file changes result in micro pattern changes.

Note that the total pattern evolution kind count (Table 5) is greater than the total class file change count (Table 1), since a class file can have more than one pattern and a class change includes more than one pattern evolution kind. The multiplicity of micro patterns are explained in [7].

Table 5. Ratio of pattern evolution kinds the three projects.

	ArgoUML	Columba	jEdit
Pattern unchanged	5,238 (94%)	12,977 (95%)	7,403 (95.9%)
Pattern changes	313 (6%)	643 (5%)	287(4.1%)

4.3 Bug-prone Pattern Evolution Kinds

We count bug-introducing changes for each micro pattern evolution kind, and compute the bug change rate for each kind. After computing all bug introduction rates for all pattern evolution kinds, we draw contour graphs to indicate the common bug-prone pattern evolution kinds. Figure 2 shows the bug introduction rates for each micro pattern evolution kind for ArgoUML. The x-axis indicates to-patterns and y-axis indicate from-patterns. For example, the left-bottom cross indicates the bug rate of the 'NONE → NONE' pattern evolution kind. The order of micro patterns along the x-axis and y-axis is the same as the ordering in Table 3, excluding the infrequently occurring * marked patterns. The contour line density indicates the bug rates. Note that the value associated with each contour line varies by chart, since each chart scales the contours to improve presentation. Contour graphs show the overview properties of bug-prone pattern evolution kinds. Denser contour lines indicate higher bug rates.

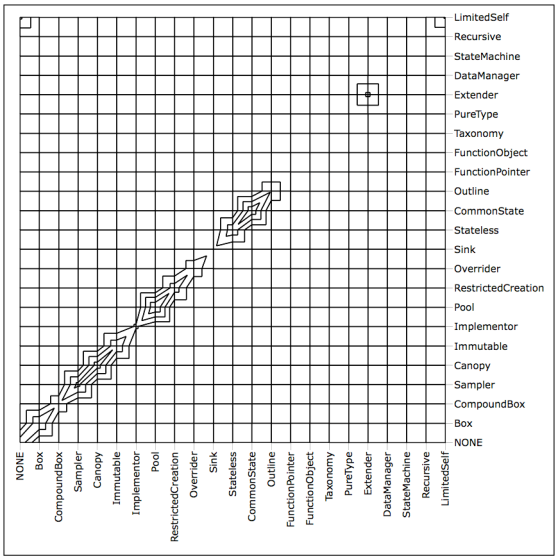


Figure 2. ArgoUML bug-prone pattern evolution kinds

Figure 3 shows bug introduction rates for each micro pattern evolution kind for jEdit. The two contour graphs (Figure 2, and Figure 3) show that the bug-prone micro pattern evolution kinds of the two projects are somewhat similar, but not identical. For example, the 'Sampler → Sampler' micro pattern evolution kind is bug-prone in all projects. However, the 'CompoundBox → Canopy' micro pattern evolution kind is bug-prone in jEdit, but not in ArgoUML. Table 6 shows the top 20 most bug-prone pattern evolution kinds of all three projects.

Table 6. Top 20 most bug-prone pattern evolution kinds.

Rank	ArgoUML Pattern evolution kind	bug rate	Columba Pattern evolution kinds	bug rate	jEdit Pattern evolution kinds	bug rate
1	Pool → Pool	40	Implementor → NONE	26	Sampler → Sampler	72
2	CommonState → CommonState	37	RestrictedCreation → RestrictedCreation	22	Recursive → Recursive	63
3	Canopy → Canopy	36	CompoundBox → CompoundBox	21	CommonState → CommonState	58
4	Sampler → Sampler	33	Immutable → Immutable	21	FunctionObject → NONE	53
5	Box → Box	30	CobolLike → NONE	20	CompoundBox → CompoundBox	52
6	Immutable → Immutable	29	NONE → Implementor	19	LimitedSelf → LimitedSelf	51
7	NONE → NONE	27	NONE → NONE	18	NONE → NONE	49
8	Stateless → Stateless	27	LimitedSelf → NONE	18	Pool → Pool	48
9	RestrictedCreation → RestrictedCreation	26	Recursive → Recursive	18	Immutable → Immutable	43
10	Extender → Extender	23	Overrider → NONE	17	Outline → Outline	42
11	LimitedSelf → NONE	22	NONE → Extender	17	Immutable → NONE	40
12	LimitedSelf → LimitedSelf	21	NONE → Overrider	15	Implementor → NONE	38
13	Outline → Outline	19	CommonState → CommonState	14	Sink → NONE	38
14	NONE → CobolLike	18	FunctionObject → FunctionObject	13	NONE → LimitedSelf	36
15	CompoundBox → CompoundBox	17	Box → Box	13	Stateless → Stateless	34
16	Overrider → Overider	17	Stateless → Stateless	13	CompoundBox → NONE	33
17	Implementor → Implementor	11	Extender → NONE	13	PureType → PureType	32
18	NONE → CommonState	10	Extender → Extender	12	Implementor → Implementor	32
19	NONE → FunctionObject	10	Canopy → Canopy	12	Extender → Extender	31
20	Stateless → RestrictedCreation	9.1	CobolLike → CobolLike	12	Overrider → Overrider	31

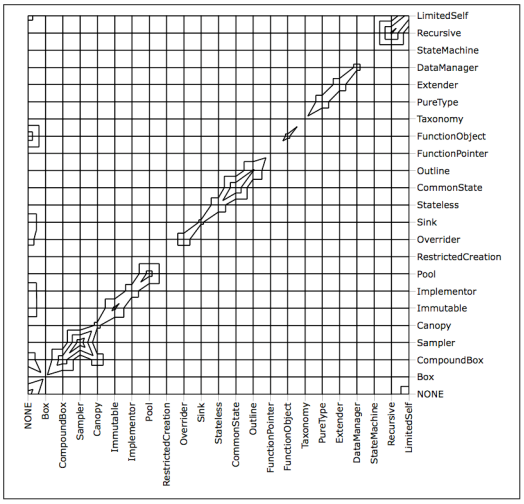


Figure 3. jEdit (rev 1-1449) bug-prone pattern evolution kinds

We observe bug-prone micro pattern evolution kinds in two different periods of the same project, jEdit. The bug rates of each micro pattern evolution kind of the two periods are shown in Figure 4 (revisions 1-500) and Figure 3 (revisions 1-1449). The bug introduction rates of the two periods are almost identical. We conclude that the bug rates of micro pattern evolution kinds of different projects are typically, but not always, similar. Bug-prone pattern evolution kinds from two different periods of the same project are very similar. We expect that quality assurance personnel could use already observed bug-prone micro pattern evolution kinds in a project to predict future bug-prone pattern evolution kinds for that same project.

4.4 Frequencies, Pattern Evolution Kinds, and Bug Rates

Since the Java class changes that cause pattern changes are infrequent (only 4 to 6% in Table 5), in this section we observe only class file changes that do not change micro patterns, such as 'NONE → NONE', 'Box → Box', and Sampler → Sampler.' We compare the frequencies, the number of the evolution kinds, and bug rates of these micro patterns. To permit cross-project comparison, we normalize each value (i.e., frequency, the number of the evolution kinds, and bug rate) by dividing each value by the sum of the values. For example, each change count is divided by the total number of changes to compute a normalized change count. The sum of normalized values is 1. The normalized values show the distribution of values among micro patterns. Figure 5-Figure 7 show the normalized values of each pattern of the three projects. For example, in Figure 5, 35% of the micro pattern evolution kinds are 'NONE → NONE,' as shown in the middle bar for 'NONE' (this is slightly higher than the 33% value for NONE → NONE in Table 4, since we have eliminated rates of class file changes that change micro patterns, and then recomputed frequencies). However, the bug introduction rate of 'NONE → NONE' is relatively low. Though Table 6 indicates that 27% of these transitions are buggy, they are only 6% of total project bugs. In contrast, the 'CommonState→CommonState' transition is found in only 1.6% of changes (see Table 4), but it contributes 9% of total project bugs. Clearly this is a dangerous type of change.

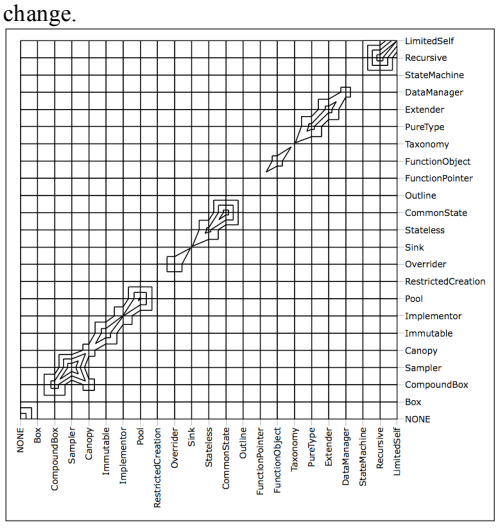


Figure 4. jEdit (rev. 1-500) bug-prone pattern evolution kinds

We observe that, in general, the fact that a pattern frequently occurs in the source code does not necessarily mean that it frequently changes. Similarly, the number of pattern changes and the bug introduction rate are not strongly correlated. Some patterns have many changes, but low bug introduction rates. There are common patterns, which occur less frequently and have small change numbers, but high bug introduction rates. For example, the 'Box', 'CompoundBox', 'Pool', 'CommonState', and 'Outline' micro patterns have high bug rates, but their frequencies and change counts are low. In contrast to that, 'Overrider' and 'Sink' micro patterns have comparatively lower bug rates.

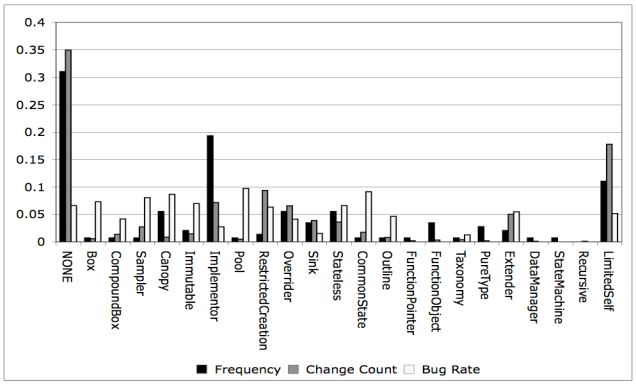


Figure 5. Micro pattern distributions, the number of changes, and bug rates of ArgoUML

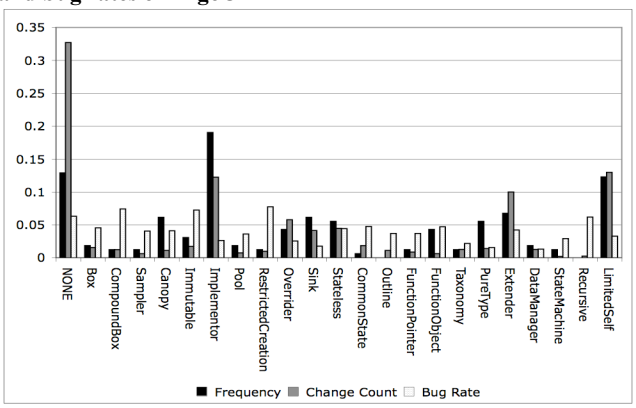


Figure 6. Micro pattern distributions, the number of changes, and bug rates of Columba

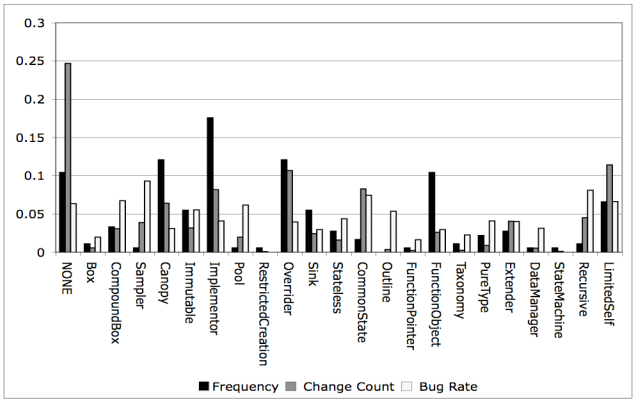


Figure 7. Micro pattern distributions, the number of changes, and bug rates of jEdit

5. DISCUSSION

5.1 Generalization

We identified the common pattern evolution kinds and bug-prone micro patterns of three projects. We showed that the three projects share some common properties, but they are not identical. We analyzed only these three projects, so it is hard to determine if our findings are broadly generalizable.

However, we showed that the bug-prone pattern evolution kinds of two different periods of the same project are very similar. This indicates that the common pattern evolution kinds and bug-prone micro patterns discovered in part of a project's history can be generalized for the remainder of the project's history.

5.2 Bug-prone Patterns

We identified common bug-prone micro patterns of the three analyzed projects, and summarize our findings in Table 7. Why are some micro patterns more bug-prone than others? Understanding bug-prone micro patterns may lead to a deeper understanding of the causes of bug-introducing changes. We also note that, since a class changes micro patterns so infrequently, most of our results are really noting correlations between individual micro patterns and bug-proneness, and not correlations between changes of micro pattern and being bug-prone.

Table 7. More/less bug-prone micro patterns

Category	Micro Pattern evolution kinds/micro patterns
Bug-prone Pattern evolution kinds	Pool → Pool, Implementor → NONE, Sampler → Sampler, CommonState → CommonState, Canopy→Canopy, Recursive → Recursive
High bug rate patterns	Box, CompoundBox, Sampler, Pool, Outline, CommonState
Low bug rate patterns	Overrider, Sink

Identifying pattern specific bugs may provide insight into the causes of bug creation. However, since identifying micro pattern specific bugs requires manual project analysis, it is very labor-intensive. In our limited explorations to date, we have not found strong examples or trends in pattern-specific bugs. Identifying trends in micro pattern specific bugs remains as future work.

5.3 Threats to Validity

There are four major threats to the validity of this work.

Systems examined might not be representative. We examined three systems. It is possible that we accidentally chose systems that have similar (or different) micro design patterns and evolution properties. Since we intentionally only chose systems that had some degree of linkage between change tracking systems and the text in the change log (so we could determine bug-introducing changes), we have a project selection bias. It certainly would be nice to have a larger dataset.

Systems are all open source and written in Java. The systems examined in this paper all use an open source development methodology and are written in Java, and hence might not be representative of all development contexts. It is possible that the stronger deadline pressure of commercial development could lead to different micro pattern change properties.

Some revisions are not compilable. To extract micro patterns from Java source code, we need to compile them and create class files

first. Analyzed open source projects contain revisions that cannot be compiled, with reasons ranging from syntax errors to missing library files. We skipped non-compilable source code, which may affect the results.

Bug-introducing change data is incomplete. We rely on the change logs to identify bug-introducing changes. Even though we selected projects that have good quality change logs, we still are only able to extract a subset of the total number of bugs. The bug change identification relies on the heuristic algorithm given in [13], so it may have false positives and false negatives.

6. RELATED WORK

Patterns in software design and implementation have been explored by many research efforts. In object-oriented designs, design patterns describe the relationships and interactions between classes or class instances and the template to manage them. In [6], Gamma et al. discussed some design patterns that are categorized into creational patterns, structural patterns, and behavioral patterns. Heuzeroth et al. [8] explored automatic design pattern detection in legacy code using static and dynamic analyses, in which patterns like Observer, Composite, Mediator, etc. are identified from Java code. In [12], Prechelt et al. presented a system called Pal that discovers structural design patterns in C++ software by examining the C++ header files. Livshits and Zimmermann combined software repository mining and dynamic analysis to discover common usage patterns and code patterns that likely encounter violations in Java applications [10]. Code-Web [11] discovers library reuse patterns in the ET++ application framework through data mining. Micro patterns are at an abstract level between design patterns and implementation patterns. Compared to design patterns, micro patterns are extractable; compared to implementation patterns that need static or dynamic analysis to discover them, micro patterns require less computation to extract.

Gil and Maman perform analysis on the prevalence of micro patterns across the Sun JDK versions 1.1, 1.2, 1.4, 1.4 and 1.4.2. They only compared distributions in each release and conclude that pattern prevalence tends to be the same in software collections [7]. We analyzed not only distributions, but also pattern evolution kinds and bug-prone change kinds.

Signature change pattern analysis [9] is similar to ours in that they try to observe signature change patterns over revisions. However, they observed only signatures change patterns, while our approach analyzes micro pattern evolution, which includes non-trivial idioms of each Java class. We also identify bug-prone patterns among identified patterns.

7. CONCLUSIONS AND FUTURE WORK

We observed the micro pattern evolution properties of three open source projects, including frequencies of micro patterns, common micro pattern evolution kinds, and bug-prone micro patterns. We found that the micro pattern distributions and common change kinds of analyzed projects are similar. The bug rates of patterns of different projects are somewhat similar. However, the bug rates of two different periods of the same projects are almost identical. We conclude that the identified bug-prone patterns from a part of a project history can be used to predict or raise awareness of the future pattern changes for the project.

We need to analyze more software projects to see if our findings can be generalized to other projects. The micro patterns are not originally designed to identify more/less bug-prone modules. We need to mine or develop new patterns to easily identify more/less bug-prone patterns. In addition, we need to mine finer granularity patterns for use at the function/method level. The software pattern evolution analysis methodology used in this paper can be reusable for other software patterns.

8. ACKNOWLEDGMENTS

Our thanks to Itay Maman and Joseph (Yossi) Gil for allowing us use the micro pattern extraction tool and for their valuable feedback.

9. REFERENCES

[1] B. Behlendorf, C. M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman, "Subversion Project Homepage," 2005, http://subversion.tigris.org/.

[2] B. Berliner, "CVS II: Parallelizing Software Development," Proc. Winter 1990 USENIX Conf., Washington, DC, pp. 341-351, 1990.

[3] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[4] J. W. Cooper, *The Design Patterns: Java Companion*: Addison-Wesley, 1998.

[5] R. E. Courtney and D. A. Gustafson, "Shotgun Correlations in Software Measures," *Software Engineering J.*, v. 8, pp. 5 - 13, 1992.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[7] J. Y. Gil and I. Maman, "Micro Patterns in Java Code," proceedings of the 20th Object Oriented Programming Systems Languages and Applications, San Diego, CA, USA, pp. 97 - 116, 2005.

[8] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic Design Pattern Detection," Proc. 11th IEEE Int'l Workshop on Program Comprehension, pp. 94, 2003.

[9] S. Kim, E. J. Whitehead, Jr., and J. Bevan, "Analysis of Signature Change Patterns," Proc. Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, MO, USA, pp. 64-68, 2005.

[10] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," Proc. 2005 European Software Engineering Conf. and Foundations of Software Eng. (ESEC/FSE 2005), Lisbon, Portugal, pp. 296-305, 2005.

[11] A. Michail, "Data Mining Library Reuse Patterns in User-Selected Applications," Proc. 14th International Conference on Automated Software Engineering, Cocoa Beach, Florida, USA, pp. 24–33, 1999.

[12] L. Prechelt and C. Krämer, "Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns," *J. Universal Computer Science*, vol. 4, pp. 866-882, 1998.

[13] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, MO, USA, pp. 24-28, 2005.

Mining Sequences of Changed-files from Version Histories

Huzefa Kagdi, Shehnaaz Yusuf, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242

{hkagdi, sdawoodi, jmaletic}@cs.kent.edu

ABSTRACT
Modern source-control systems, such as *Subversion*, preserve change-sets of files as atomic commits. However, the specific ordering information in which files were changed is typically not found in these source-code repositories. In this paper, a set of heuristics for grouping change-sets (i.e., log-entries) found in source-code repositories is presented. Given such groups of change-sets, sequences of files that frequently change together are uncovered. This approach not only gives the (unordered) sets of files but supplements them with (partial temporal) ordering information. The technique is demonstrated on a subset of *KDE* source-code repository. The results show that the approach is able to find sequences of changed-files.

Categories and Subject Descriptors
D.2.7. [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation, enhancement*, *extensibility, version control*

General Terms
Management, Experimentation

Keywords
Mining Software Repositories, Heuristics, Change Sequences

1. INTRODUCTION
Source-code repositories store metadata such as user-ids, timestamps, and commit comments. This metadata explains the *why*, *who*, and *when* dimensions of a source-code change. Researchers have utilized this type of information for a variety of purposes in the context of supporting and understanding software evolution [5, 6, 9, 12-15] . This includes discovering entities (e.g., files) that frequently change together for the purpose of supporting software-change prediction [3, 8, 10, 11, 16, 17, 21]. Software-change prediction approaches based on itemset mining produce unordered collections of the changed entities. For example, a set of files {f1, f2} that are frequently changed or rules such as changes in a set {f1, f2} leads to changes in a set {f3, f4}.

MSR'06, May 22–23, 2006, Shanghai, China.

However, software changes are inherently (partially) ordered along the time dimension[1]. Itemset mining approaches ignore the ordering information in the mining phase. However, the ordering must be considered at a later stage in software-change prediction. For example if a set of changed-files {f1, f2} is equivalent to a set {interface, implementation}, the changes are not necessarily symmetric. The mined set {f1, f2} may be an artifact of only the interface changes, {f1} leading to implementation changes, {f2} and not vice-versa. Therefore, ignoring the ordering information could lead to a false prediction of {f1} due to a change in {f2}.

Here, we explore the ordering of changed-files by utilizing the information found in the versions log of source-code repositories. We present an approach that processes the log-entries to deduce the partial ordering information among changed-files. For example, our approach discovers sequences of changed-files such as {f1}→{f2} and {f4}→{f5}. The sequence {f1}→ {f2} indicates that changes in {f1} *happens before* {f2}. We term this problem as mining sequences of changed-files. We define six heuristics for grouping the log-entries (i.e., change-sets) of a source-code repository. Given such a group of log-entries we uncover sequences of files that frequently change together. This approach gives not only the (unordered) sets of files but supplements them with (partial) ordering information. Therefore, this approach of changed-files sequence-mining subsumes the approach of changed-files itemset mining.

The rest of the paper is organized as follows. In section 2, we discuss the available change-set records from source-code repositories. In section 3, we present heuristics for grouping. In section 3, we discuss frequent sequence mining. In section 4, we describe the developed toolset. In section 5, we apply our approach on *KDE* version history. In section 6, we briefly discuss related work. Finally, we state our conclusions and future directions in section 7.

2. CHANGE-SETS RECORDS
There is an inherent temporal ordering between various change-sets. It is not uncommon to have a change-set either planned (e.g., a standard refactoring or a fix for a documented bug) or unplanned activity (e.g., a violation of hidden dependencies) leading to further change-sets. First, we examine how these change-sets are recorded in repositories maintained by modern source control systems.

Among several other improvements over CVS and alike, modern source-control systems, such as *Subversion*, preserve the grouping

[1] In the rest of the discussion, ordering and temporal ordering are used interchangeably unless specified.

of several changes in multiple files to a single change-set as performed by a committer (i.e., an atomic commit). Version-number assignment and metadata are associated at the change-set level and recorded as a *logentry*. As shown in Figure 1, a change-set is stored as a single logentry. *Subversion*'s log-entries include the (structured) dimensions *committer*, *date*, and *paths* (i.e., files) involved in a change-set. As shown in Figure 1, each logentry is uniquely identified by a *revision* number. There is no temporal ordering between paths *khtml_part.cpp* and *loader.h*. Clearly, the logentry alone is insufficient to give the temporal ordering of the files involved in a change-set. However, there is a temporal order between change-sets. Change-sets with greater revision numbers occur after those with lesser revision numbers. Therefore, we can utilize the ordering of change-sets to determine ordering of files.

A straightforward approach is to exhaustively list all the sequences of the changed-files. For example, if a change-set {f1, f2} occurs before {f3. f4}, the possible changed-file sequences are {f1}→{f3}, {f1, f2}→{f3}, and so forth. However, this leads to two major issues: 1) sequences that may not be useful for software evolution tasks such as change predication (i.e., false positives) and 2) examination of combinatorial explosion of changed-file sequences. Notice that the atomic commits are serialized. The temporal order in which log-entries appear in the log files is at discretion of a version-control system. As a result successive log-entries may be unrelated in the context of changes performed in the files. Therefore, it may result in meaningless changed-file sequences.

In an effort to avoid reporting of meaningless changed-file sequences, we define heuristics for grouping "related" change-sets. Furthermore, given such related change-sets, we employ sequence mining to effectively deal with the combinatorial explosion of search space.

3. CHANGE-SET GROUPING HEURISTICS

The heuristics are driven by grouping of log-entries based on the dimensions committer, date, and the paths as discussed below.

Time Interval - Change-sets committed in the same time-interval are related and change-sets committed in different time-intervals are unrelated. This helps define ordering on the change-sets in the same time-interval. Therefore, all the change-sets (i.e., log-entries) committed in a given time duration are placed in a single group. The sequences of files found using this heuristic implies that if a file is modified in a sequence on a day, the following (preceding) files are modified on the same day.

Committer – The change-sets modified by a committer are related and the change-sets modified by different committers are unrelated. This defines an order on the change-sets by a committer. Therefore, all the change-sets (i.e., log-entries) committed by a given committer are placed in a single group. The sequences of files found using this heuristic implies that if a file is modified in a sequence by a committer, the following (preceding) files are modified by the same committer.

File – Change-sets involving a particular file are related. This defines ordering on the change-sets by a particular file. Therefore, all the change-sets (i.e., log-entries) committed in which a given file is involved are placed in a single group. The sequences of files found using this heuristic establishes a temporal position of a file in the sequences of changes with other files.

Joins of the above heuristics lead to further groupings of change-sets. For example, the join of heuristics Committer and File implies committers typically do not work on a same change-set. However, committers may work on the same file in different time intervals. This defines an order on the changed-files by a committer in a time interval. The sequences of files found using this heuristic implies that if a file is modified on a day by a committer, the following (preceding) files are modified on the same day by the same committer.

```
<?xml version="1.0" encoding="utf-8"?>
<log>
  <logentry  revision="438663">
    <author>kling</author>
    <date>2005-07-25T17:46:20.434104Z</date>
    <paths>
      <path action="M">khtml_part.cpp</path>
      <path action="M">loader.h</path>
    </paths>
    <msg>
       Do pixmap notifications when
       running ad filters.
    </msg>
  </logentry>
</log>
```

Figure 1. A Snippet of kdelibs Subversion Log

These heuristics are the first step towards mining sequences of changed-file sets. Heuristics helps us to define a logical grouping of change-sets but does not directly give the order in which files were changed within a given change-set. In the next section, we describe our approach of mining frequent sequences of changed-files from grouped change-sets.

4. MINING CHANGED-FILE SEQUENCES

The problem of mining sequences of changed-files is an instance of mining frequent sequences of items. We first give definitions of a sequence and the problem of frequent sequence mining. Then, we show the reduction of our problem to the problem of mining frequent sequences.

A frequent-sequence is made up of (ordered) elements. Each element is made up of (unordered) items. The ordering of elements imposes a partial order on the items. For example, the frequent sequence {f1, f2}→{f3, f4}→{f5} is made up of 3 elements and 5 items. It indicates that the element {f1, f2} happens before the element {f3, f4} and the element {f3, f4} happens before the element {f5}. However, the happens before relation between items f3 and f4 is unknown in the element {f3, f4}. Therefore, a frequent-sequence establishes both the ordered and unordered relationship between items.

The problem of finding frequent sets of sequences is formally defined as given a set of items, $\alpha = \{i_1, i_2, \ldots. i_m\}$, and a set of transactions, $\tau = \{T_1, T_2, \ldots.., T_n\}$, find all the sets of sequences, $S = \{S_1, S_2, \ldots S_o\}$, that co-occur in at least a given number (or percentage) of transactions i.e., it satisfies a given minimum support, σ_{min}. Each Transaction contains an ordered list of events and is identified by an unique id, $T_i = (tid, \varepsilon)$ where $\varepsilon = [E_1, E_2, \ldots., E_p] \mid \forall_{i,j}\ E_i \rightarrow E_j$. and $\rightarrow$ is a given ordering relation on events. Each event contains a set of items and is identified by an unique id, $E_i = (eid, \subseteq \alpha)$. Each sequence is defined as an ordered list of elements (i.e., itemsets), $S_I = [I_1 \rightarrow I_2 \rightarrow \ldots..\rightarrow I_p] \mid \forall I_i \subseteq \alpha$,

and each member of an element, $i_j \in I_i$ is defined as an item of a sequence. A mined sequence S_i is called a *frequent sequence.* A sequence consisting of k items is referred to as a *k-sequence*. The number (or percentage) of transactions in which a sequence occur is known as its *support* (*frequency*).

The problem of frequent-sequence mining was introduced by Agrawal [1]. A number of algorithms for frequent-sequence mining are proposed. Their discussion is out of the scope in this paper. The reduction of our problem of mining sequences of changed-files to that of frequent-sequence mining is straightforward. Here, we have a set of transactions, τ, mapped to the grouping of change-sets formed by the application of a heuristic on the log-entries, events $E_i \rightarrow \ldots \rightarrow E_j$ in each transaction, T_i, maps to the change-sets ordered by revision numbers. Following this reduction, the solution to the frequent-sequence problem, S, gives the solution to the frequent changed-files sequences.

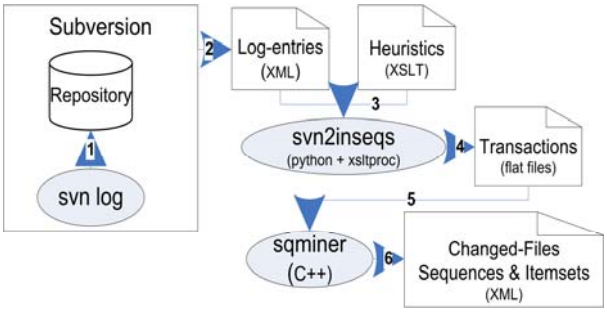


Figure 2. Tool-Chain for Mining Frequent Changed-files

5. MINING TOOLSET

We developed *svn2inseqs* and *sqminer* to process Subversion log-entries and mine the sequences of changed-file. The tool-chain is shown in Figure 2. The overall process is: 1) use the *svn log* command to produce the log-entries in XML format, 2) use *svn2inseqs* to apply a grouping heuristic on the log-entries and obtain the input transactions and events, and 3) finally, use *sqminer* to find the sequences of changed-files. In the following sections, we expand on phases 2 and 3.

5.1. Log-entries to Input-Transactions

Six grouping heuristics, *date (time interval)*, *author*[2] *(committer)*, *file*, *author-date*, *author-file*, and *date-file* are implemented as *XSLT* programs. The *Python* script *svn2inseqs* takes as input the log-entries such as the one shown in Figure 1 and a *XSLT* grouping heuristic, and transforms the log-entries to the corresponding input transactions and events in a flat-file format. One such example of input transactions obtained from the log-entries of the *kdelibs* repository is shown in Figure 3. Here, the log-entries are grouped by the heuristic author-date. The input-transactions file contains a set of events, each specified on a separate line. An event description consists of a generated input transaction-id (e.g., *14*). The transaction-id corresponds to an author-date combination i.e., the change-sets performed by an author (*giessl*) on a particular date (*2005-07-26*). The revision number of a log-entry (i.e., a change-set) is used as an event-id (e.g., *438962* and *438971*). Finally, the files involved in a change-set are listed.

```
...........
14  438962 plastik/plastik.cpp
14  438971 plastik.cpp
...
116  449301 plastik.cpp
116  449436 kstyle.cpp kstyle.h
116  449437 plastik.cpp
116  449483 kstyle.cpp kstyle.h
116  449484 plastik.cpp
116  449494 plastik.cpp
116  449521 plastik.cpp
.......
```

Figure 3. A Snippet of Input Transactions from *kdelibs* Log-entries grouped by Heuristic Author-Date.

5.2. Mining Changed-file Sequences from Input Transactions via *sqminer*

The transactions constructed by the application of heuristics such as the one shown in Figure 3 are fed to the sequence mining tool, *sqminer*. The tool *sqminer* is realized based on the Sequential Patten Discovery Algorithm (SPADE) [19]. The SPADE algorithm utilizes an efficient enumeration of sequences based on common-prefix subsequences and division of search space using equivalence classes. Additionally, it utilizes a vertical input-transaction format for an efficient counting of support values. The configuration parameters of *sqminer* include support, max number of items allowed in a sequence, mining of sequence (association) rules, and output in both the flat-file and XML format.

```
<frequent-sequences input="kdelibs-authordate">
<frequent-sequence size="2" support="6">
  <elements>
    <element temporal-position="1">
      <items><item>plastik.cpp</item></items>
    </element>
    <element temporal-position="2">
       <items><item>plastik.cpp</item></items>
    </element>
  </elements>
  <idpairs>
    <idpair seqid="14" eventid="438971"/>
    <idpair seqid="116" eventid="449437"/>
    <idpair seqid="116" eventid="449484"/>
  </idpairs>
</frequent-sequence>
</frequent-sequences>
```

Figure 4. A Snippet of Sequences of Changed-files from *kdelibs* Log-entries grouped by Heuristic Author-Date.

Figure 4 shows an example of a sequence of changed-paths mined from *kdelib* with a minimum-support value of 3. In this case, a file (*plastik.cpp*) was changed twice in a sequence. Transactions formed from (possibly different) author-date values that have common files in their change-set(s), contribute to the support value of a sequence of changed-files. In Figure 3, transactions *14* and *116* are not formed from the same author-date values but both support the sequence $\{plastik.cpp\} \rightarrow \{plastik.cpp\}$. In addition to the number of elements (i.e., *size*) and support values, information about the transactions (*seqid* and *eventid*) in which these

[2] We do not make the distinction between authors, contributors, volunteers, and so forth.

sequences are found (i.e., *idpairs*) is also stored. This provides the user (application or human) an additional context for their tasks.

Once we have a sequence, it can easily be reduced to an itemset. For example, if a sequences a→b and b→a are found to have the same support (and/or same *id-pairs*), it can be generalized to an itemset {a, b}. The relationship between itemsets and sequences is one-to-many i.e., it is possible that multiple sequences are reduced to a single itemset. For example, the sequence shown in Figure 4 is reduced to itemset as shown in Figure 5. An itemset is a single element sequence composing of all the items in the corresponding sequence. Notice that our approach produces a multi-set. This information not only gives the items that were involved but also the number of their instances implicitly.

6. EVALUATION

The presented heuristics and mining technique are evaluated on an open source system. The primary interest is to show that our approach is able to find frequent sequences of changed-files.

6.1. Dataset Acquisition

The considered version history from the *KDE Subversion* repository is shown in Table 1. The dataset was collected on the 25th of January 2006. The log-entries were collected separately for each of the *KDE* modules and the entire *KDE*. The *svn log* command was used to extract the logs in *XML* format. Table 1 shows the number of revisions, days these revisions were committed, authors, the number of (unique) files involved in the change-sets (i.e., changed files column), and the number of changes performed in these files. Note that the cumulative sum of an attribute values may not agree with the corresponding value of *KDE* due to common values across modules.

```
<frequent-sequences input="kdelibs-authordate">
<frequent-sequence size="2" support="6">
  <elements>
    <element temporal-position="1">
       <items>
         <item>plastik.cpp</item>
         <item>plastik.cpp</item>
       </items>
    </element>
  </elements>
  <idpairs>
    <idpair seqid="14" eventid="438971"/>
    <idpair seqid="116" eventid="449437"/>
  </idpairs>
</frequent-sequence>
</frequent-sequences>
```

Figure 5. A Snippet of Itemsets of Changed-files from kdelibs Log-entries grouped by Heuristic Author-Date.

6.2. Application of Heuristics

After acquiring the dataset, heuristics were applied (*Day, Author*, *File, Author-date*, *Author-file*, and *Day-file*) with the help of a tool *svn2inseqs* and the input files required by *sqminer* were generated. A calendar day is mapped to the heuristic *Time-interval*. In this study, log-entries consisting of more than ten files were pruned. This was done to discard noisy change-sets such as those updating the license information. Transactions with a single event were also ignored as there is no temporal ordering found in a singleton transaction.

The number of transactions and events obtained from the application of each of the heuristics is given in

Table 5. The transactions and events maintain the invariant |events| ≤ |Revisions| on account of pruning. The events are distributed among the transactions based on the grouping dictated by the heuristics. Therefore, the *transaction density* (i.e., number of events in a transaction) is directly dependent on the applied grouping heuristic. The above invariant implies that more the number of transactions, lower the transaction density. On the other end less the number of transactions, higher the transaction density. The maximum changed-files sequence size (i.e., number of elements) is less than or equal to the maximum number of events found in a transaction. Based on the above properties, we have the following observations: 1) transactions with high-density values are likely to find long-size sequences of changed-files with a less number of supporting transactions and 2) transactions with low-density values are likely to find small-size sequences of changed-files with a more number of supporting transactions.

Table 1. Log Information of the *KDE* Source-Code Repository

Modules	Period: (07-25-2005 to 01-25-2006)				
	Revisions	Days	Authors	Changed-Files	Changes in files
arts	3	2	2	3	4
kde-common	295	125	36	30	325
kdeaccesibility	164	71	12	3129	3684
kdeaddons	155	77	19	3085	3623
kdeadmin	91	53	13	2979	3396
kdeartwork	95	32	10	4423	4598
kdebase	1493	173	90	6027	13364
kdebindings	129	43	6	3214	3424
kdeedu	886	154	37	4360	8077
kdegames	216	71	21	3490	4444
kdegraphics	509	147	22	4032	6452
kdelibs	3557	184	124	5262	20604
kdemultimedia	185	84	24	3210	4493
kdenetwork	553	132	31	4860	9012
kdepim	1944	181	57	7522	15990
kdesdk	686	157	32	4178	6727
kdetoys	86	46	16	2917	3193
kdeutils	333	109	31	3947	5858
kdevelop	375	57	15	6965	9085
kdewebdev	19	16	6	2920	2950
KDE	11170	185	230	30648	82662

6.3. Mining Sequences of Changed-Files

sqminer was executed on the transactions of the KDE modules listed in Table 5. The mining of sequences of changed-files was performed on a number of support values. The maximum number of files (i.e., items) in a sequence was set to 20 in all the runs. These support values were selected taking into account the two observations (transaction density) mentioned in the previous section (6.2). Here, the discussion and analysis of the results are limited to a subset of the results. Our intention is to show the distribution of the sequences found in the context of various tool configurations.

Table 6 shows the sequences of changed-files found from the transactions corresponding to the heuristics used in the mining process. The number of sequences (S) found with a configuration

of the minimum support (σ_{min}), the maximum sequence size ($|E_m|$) along with the maximum number of files ($|\alpha_m|$), and the run-time (T) are presented. Sequences were found for a range of minimum support values (σ_{min}). The heuristics Day and Day-Author found sequences in all the cases. The heuristic Author did not report sequences in three cases. The heuristic File and Author-File did not report sequences in six cases. Finally, the heuristic Day-File did not report any sequences in seven cases. There were no sequences found for *arts, kdewebdev*, and *kdeartwork*.

Table 2. Sequences in *kdelibs*

kdelibs	Sequences with no of Elements									
	1	**2**	**3s**	**4**	**5**	**6**	**7**	**8**	**|S|**	**$\|\alpha_m\|$**
Day (3)	900	1304	593	58	1	-	-	-	2856	7
Author(3)	469	2079	3866	4401	2919	1214	272	18	15232	8
File (15)	219	98	23	-	-	-	-	-	340	5
Day-Author(3)	717	274	36	-	-	-	-	-	1027	5
Day-File (15)	44	26		-	-	-	-	-	70	4
Author-File (15)	157	100	10	-	-	-	-	-	267	5

Table 3. Comparison of Itemsets and Sequences in *kdelibs*

	Itemset		**Sequences**		**Ratio**
kdelibs	**|I|**	**$\|\alpha_m\|$**	**|S|**	**$\|\alpha_m\|$**	**|S|/|I|**
Day (3)	2193	7	2856	7	1.3
Author(3)	9575	8	15232	8	1.6
File (15)	263	5	340	5	1.3
Day-Author(3)	907	5	1027	5	1.13
Day-File (15)	55	4	70	4	1.27
Author-File (15)	190	5	267	5	1.41

The results presented in Table 6 shows that sequences of changed-files are found from the log-entries using our approach. However, it remains to be seen that the supplementary information in sequences is useful. To facilitate the discussion, *kdelibs* is taken as a representative. Table 2 shows the sequences of changed-files found by applying each of the six heuristics and the min-support value (e.g., 3). In case of the heuristic Author, more than 15,000 sequences were reported. Further, the distribution of these sequences based on the number of elements it contains is also shown. The maximum number of elements found in sequences was reported to be 8. The maximum number of files found in sequences was also reported to be 8.

These results give interesting insights into the software evolution process used by *kdelib* authors. The long sequences of changed-files under the heuristic Author indicate that related changes are committed in small increments spreading across multiple change-sets (revisions). Furthermore, examining the results under the heuristics Day-Author, the sequences are fewer and relatively smaller in size. This means that related changes are typically not committed on the same day and by the same author. This information leads to the hypothesis that related changes for a high-level modification (e.g., feature or bug-fix) are performed in incremental steps. Even if this hypothesis is not verified, we at least have the historical dependencies between high-level changes.

Moreover, the importance of ordering can be seen for tasks such as assisting a developer with the software-change process. For example, if a sequence consisting of eight elements is treated as an itemset, number of candidates of the combinatorial order may need to be considered. This may lead to lack of precision as many of these combinations may turn-out to be false-positives.

Based on the prior discussion, we provided a case for sequences giving more concise information for tasks such as software-change prediction. However, there is a possibility of a large number of sequences (combinatorial order) being produced compared to itemsets. Our approach facilitates decision-making on this issue. The sequences are also reduced to itemsets and output by *sqminer*. Table 3 facilitates the comparison of sequences and itemsets of changed-files for *kdelibs*. In this case, the number of sequences is less than twice the number of itemsets. Each itemset is essentially ordered. Therefore, our approach serves both the generalization (itemsets) and specialization (sequences) cases.

Table 4. Example of Sequences of Changed-Files in *kdelibs*

kdelibs	**Sequences**
Day (3)	{range.h} → {katedocument.cpp , katedocument.h}
	{KDE4PORTING.html } → { kstringhandler.cpp, kstringhandler.h }
Author (3)	{generic.py} → {openssl.py}
Day-Author (3)	{kstyle.h} → {plastik.cpp} → {kstyle.cpp}
	{ kateregression.cpp } → { kateregression.cpp } → { range.cpp }

We conclude this section, with examples of sequences found in *kdelibs* as shown in Table 4. The sequence shown in the heuristics *Day* is partially ordered. The file *range.h* changed before *katedocument.cpp* and *kdatedocument.h.* However, the order of changes between *katedocument.cpp* and *kdatedocument.*h is undecided. The sequence in the *Day-Author* demonstrates the ordering between an interface file (*kstyle.h*) and an implementation file (*kstyle.cpp*).

7. RELATED WORK

We briefly discuss approaches utilizing information found in source-code repositories maintained by tools such as *CVS* and *Subversion* with a focus on software changes.

Zimmerman et al [20, 21] used *CVS* logs for detecting evolutionary coupling between source-code entities. They employed sliding window heuristics to estimate the atomic commits (change-sets). Association-rules based on itemset mining were formed from the change-sets and used for change-prediction. Yang et al [18] used a similar technique for identifying files that frequently change together. Gall et al [8] used window-based heuristics on *CVS* logs for uncovering logical couplings and change patterns, and German et al [9] for studying characteristics of different types of changes. Hassan et al [11] analyzed *CVS* logs for software-change prediction.

Van Rysselberghe et al [16] utilized *CVS* logs in their approach to find frequently applied changes and presented a 2D visualization technique to help recognize change-relevant information [17]. Bieman et al [3] used logs from software repositories to assist in

the computation of metrics for detecting change-prone classes. Burch et al [4] presented a tool that supports visualization of association rules and sequence rules. However, a very little information is provided on how *CVS* transactions are processed and sequences are mined. Beyer et al [2].used the log information in visualizing clusters of frequently occurring co-changes. Dinh-Trong et al [6] used *CVS* logs for validating previously developed hypotheses on successful open source development. Chen et al [5] incorporated the *CVS* commit messages in their source-code search tool. El-Ramly et al [7] used sequence mining to detect patterns of user activities from the system-user interaction data.

Table 5. Transactions formed by Application of Grouping Heuristics on the Log Information of the *KDE* Source-Code Repository

Modules	(No. of Transactions, No. of Events)					
	Day	Author	File	Day-Author	Day-File	Author-File
arts	(1, 2)	(1,2)	(0, 0)	(1, 2)	(0, 0)	(0, 0)
kde-common	(72, 242)	(8, 267)	(11, 38)	(64, 168)	(5, 10)	(9, 26)
kdeaccesibility	(35 ,111)	(6, 137)	(63, 212)	(31, 93)	(12, 27)	(57, 170)
kdeaddons	(31, 104)	(8, 133)	(52 , 173)	(29, 94)	(18, 36)	(44, 150)
kdeadmin	(18, 51)	(6, 73)	(33, 84)	(16, 46)	(9, 18)	(26, 60)
kdeartwork	(13, 68)	(5, 80)	(17, 40)	(13, 64)	(10, 22)	(15, 35)
kdebase	(157, 1348)	(63, 1337)	(484, 1501)	(235, 1049)	(119, 256)	(349, 879)
kdebindings	(21, 94)	(4, 111)	(17, 121)	(21, 86)	(18, 64)	(20, 118)
kdeedu	(123, 752)	(25, 760)	(344, 1371)	(164, 626)	(168, 424)	(331, 1104)
kdegames	(32, 156)	(9, 178)	(54, 154)	(30, 139)	(8, 16)	(46, 104)
kdegraphics	(109, 427)	(14, 453)	(131, 520)	(96, 335)	(41, 95)	(122, 429)
kdelibs	(182, 3304)	(89, 3271)	(911, 3757)	(605, 2720)	(420, 977)	(719, 2281)
kdemultimedia	(37, 116)	(13, 146)	(55, 175)	(37, 100)	(8, 16)	(54, 128)
kdenetwork	(95, 456)	(25, 482)	(194, 648)	(104, 390)	(70, 146)	(152, 464)
kdepim	(166, 1668)	(46, 1671)	(686, 2186)	(330, 1365)	(203, 444)	(557, 1563)
kdesdk	(125, 618)	(19, 637)	(145, 486)	(141, 517)	(53, 121)	(132, 427)
kdetoys	(20, 58)	(6, 70)	(42, 132)	(17, 49)	(14, 28)	(36, 115)
kdeutils	(63, 251)	(22, 281)	(126, 437)	(66, 212)	(55, 126)	(106, 344)
kdevelop	(28, 294)	(9, 309)	(177, 713)	(32, 282)	(170, 592)	(170, 676)
kdewebdev	(2, 4)	(3, 12)	(6, 12)	(2, 4)	(0, 0)	(0, 0)
KDE	(185, 10092)	(183, 10047)	(3373, 12142)	(1659,8753)	(1274, 2954)	(2773, 8496)

Table 6. Sequences found from the Log Information of the *KDE* Source-Code Repository

Modules	|S|– No. of Sequences, $\|E_m\|$ ($\|\alpha_m\|$)- Max Element-Size (Max) Item-size, and σ_{min}. used in Mining, T – Run-time in Seconds																	
	Day			Author			File			Day-Author			Day-File			Author-File		
	|S| (σmin)	$\|E_m\|$	T.	|S| (σmin)	$\|E_m\|$	T.	|S| (σmin)	$\|E_m\|$	T.	|S| (σmin)	$\|E_m\|$	T.	|S| (σmin)	$\|E_m\|$	T.	|S| (σmin)	$\|E_m\|$	T.
kde-common	30(3)	3(3)	<1	223(3)	6(6)	6	0	-	-	15(3)	2(2)	<1	0	-	-	0	-	-
kdeaccesibility	22(3)	2(2)	<1	1(3)	1(1)	<1	44(10)	1(3)	<1	12(3)	2(2)	<1	0	-	-	19(10)	1(2)	<1
kdeaddons	14(3)	1(7)	<1	1(3)	1(1)	<1	-	-		137(3)	1(7)	<1	31(10)	1(5)	<1	0	-	-
kdeadmin	5(3)	1(1)	<1	0	-	-	1(10)	1(1)	<1	5(3)	1(1)	<1	0	-	-	0	-	-
kdebase	309(3)	2(2)	8	216(3)	3(4)	35	3(25)	1(1)	<1	193(3)	2(2)	1	1(10)	1(1)	<1	0	-	-
kdebindings	31(3)	2(3)	<1	0	-	-	0	-	-	27(3)	2(3)	<1	4(10)	1(1)	<1	2(10)	1(1)	<1
kdeedu	518(3)	3(4)	3	1152(3)	5(6)	7	11(25)	2(2)	<1	390(3)	3(3)	1	4(25)	1(1)	<1	11(25)	2(2)	<1
kdegames	12(3)	2(2)	<1	6(3)	2(2)	<1	5(10)	1(2)	<1	8(3)	2(2)	<1	0	-	-	1(10)	1(1)	<1
kdegraphics	138(3)	3(3)	<1	55(3)	3(4)	1	0	-	-	103(3)	2(4)	<1	4(10)	1(1)	<1	1(25)	1(1)	<1
kdelibs	2856(3)	5(5)	427	15232(3)	8(8)	2247	39(25)	2(2)	10	1027(3)	3(4)	23	19(25)	2(3)	<1	36(25)	2(2)	2
kdemultimedia	19(3)	1(2)	<1	35(3)	4(4)	<1	17(10)	1(3)	<1	17(3)	1(1)	<1	0	-	-	23(10)	1(4)	<1
kdenetwork	215(3)	2(5)	1	67(3)	2(2)	<1	0	-	-	186(3)	2(5)	1	163(10)	2(7)	<1	0	-	-
kdepim	770(3)	3(3)	17	749(3)	6(7)	20	7(25)	1(1)	<1	634(3)	2(4)	3	0	-	-	4(25)	1(1)	<1
kdesdk	102(3)	3(3)	<1	12(3)	2(3)	<1	0	-	-	79(3)	3(3)	<1	2(10)	1(1)	<1	0	-	-
kdetoys	9(3)	1(2)	<1	0	-	-	1528(10)	2(9)	2	7(3)	1(1)	<1	0	-	-	1527(10)	2(9)	4
kdeutils	87(3)	3(3)	<1	20(3)	2(2)	<1	1(25)	1(1)	<1	70(3)	3(3)	<1	35(10)	2(3)	<1	75(10)	3(4)	1
kdevelop	88(3)	3(4)	<1	2(3)	1(1)	<1	2(25)	2(2)	1	82(3)	3(4)	<1	2(25)	2(2)	1	2(25)	2(2)	1
KDE	119(10)	2(3)	13	14(10)	2(2)	2	61(25)	2(2)	15	4301(3)	3(4)	350	23(25)	2(3)	1	52(25)	2(2)	4

8. CONCLUSIONS AND FUTURE WORK

We investigated the problem of mining ordered sequences of changed-files from the change-sets found in source-code repositories. Six heuristics were examined to form input transactions with ordered change-sets. A toolset was developed to uncover the sequences of changed-files from the change-sets. A case-study on the *KDE* project shows that our approach is able to find ordered sequences of changed-files ranging from none to thousands. In our experience, the number of sequences is found to be closely bounded to the number of itemsets. We formed a hypothesis that sequences can be used to better predict and analyze the software evolutionary process.

In future, we plan to assess the effectiveness of the sequences of changed-files for the task of software-change prediction. Sequences of source-code entities such as classes, methods, statements, and expressions will be analyzed and compared. Also, we plan to integrate grouping heuristics based on the textual contents of comments in the change-sets and other repositories (e.g., *Bugzilla*).

9. REFERENCES

[1] Agrawal, R. and Srikant, R. Mining Sequential Patterns in Proceedings of Eleventh International Conference on Data Engineering (Taipei, Taiwan, March, 1995).

[2] Beyer, D. and Noack, A. Clustering Software Artifacts Based on Frequent Common Changes in Proceedings of 13th International Workshop on Program Comprehension (IWPC'05) (St. Louis, Missouri, USA, May 15-16, 2005), 259-268.

[3] Bieman, J. M., Andrews, A. A., and Yang, H. J. Understanding Change-Proneness in OO Software Through Visualization in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03) (2003), 44-53.

[4] Burch, M., Diehl, S., and Weißgerber, P. Visual Data Mining in Software Archives in Proceedings of Proceedings of the 2005 ACM symposium on Software visualization (St. Louis, Missouri, May 14-15, 2005), 37-46.

[5] Chen, A., Chou, E., Wong, J., Yao, A. Y., Zhang, Q., Zhang, S., and Michail, A. CVSSearch: Searching through Source Code using CVS Comments in Proceedings of Proceedings IEEE International Conference on Software Maintenance (ICSM'01) (2001), 364-373.

[6] Dinh-Trong, T. T. and Bieman, J. M. The FreeBSD Project: a Replication Case Study of Open Source Development. IEEE Transactions on Software Engineering, 31, 6 (2005), 481-494.

[7] El-Ramly, M. and Stroulia, E. Mining Software Usage Data in Proceedings of International Workshop on Mining Software Repositories (MSR'04) (2004), 64-8.

[8] Gall, H., Hajek, K., and Jazayeri, M. Detection of Logical Coupling based on Product Release History in Proceedings of International Conference on Software Maintenance (ICSM'98) (1998), 190-199.

[9] German, D. M. An Empirical Study of Fine-Grained Software Modifications in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (2004), 316-25.

[10] German, D. M. Mining CVS Repositories, the SoftChange Experience in Proceedings of International Workshop on Mining Software Repositories (MSR'04) (2004), 17-21.

[11] Hassan, A. E. and Holt, R. C. Predicting Change Propagation in Software Systems in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (2004), 284-93.

[12] Huang, S.-K. and Liu, K.-m. Mining Version Histories to Verify the Learning Process of Legitimate Peripheral Participants in Proceedings of International Workshop on Mining Software Repositories (MSR'05) (St. Louis, Missouri, May 17, 2005), 84-78.

[13] Lopez-Fernandez, L., Robles, G., and Gonzalez-Barahona, J. M. Applying Social Network Analysis to the Information in CVS Repositories in Proceedings of International Workshop on Mining Software Repositories (MSR'04) (May 25, 2004), 101-105.

[14] Mockus, A., Fielding, T., and Herbsleb, D. Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology (TOSEM), 11, 3 (July 2002 2002), 309-346.

[15] Tu, Q. and Godfrey, M. W. An Integrated Approach for Studying Architectural Evolution in Proceedings of 10th International Workshop on Program Comprehension (IWPC'02) (2002), 127-136.

[16] Van Rysselberghe, F. and Demeyer, S. Mining Version Control Systems for FACs (Frequently Applied Changes) in Proceedings of International Workshop on Mining Software Repositories (MSR'04) (May 25, 2004), 48-52.

[17] Van Rysselberghe, F. and Demeyer, S. Studying Software Evolution Information By Visualizing the Change History in Proceedings of 20th IEEE International Conference on Software Maintenance (2004), 328-37.

[18] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. Predicting Source Code Changes by Mining Change History. IEEE Transactions on Software Engineering, 30, 9 (September 2004), 574 - 586.

[19] Zaki, M. J. SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning, 42, 1-2 (January 2001), 31 - 60.

[20] Zimmermann, T., Weibgerber, P., Diehl, S., and Zeller, A. Mining version histories to guide software changes in Proceedings of 26th International Conference on Software Engineering (2004), 563-72.

[21] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. Mining Version Histories to Guide Software Changes. IEEE Transactions on Software Engineering, 31, 6 (2005), 429-445.

MAPO: Mining API Usages from Open Source Repositories

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Jian Pei
School of Computing Science
Simon Fraser University
Burnaby, BC Canada V5A 1S6
jpei@cs.sfu.ca

ABSTRACT

To improve software productivity, when constructing new software systems, developers often reuse existing class libraries or frameworks by invoking their APIs. Those APIs, however, are often complex and not well documented, posing barriers for developers to use them in new client code. To get familiar with how those APIs are used, developers may search the Web using a general search engine to find relevant documents or code examples. Developers can also use a source code search engine to search open source repositories for source files that use the same APIs. Nevertheless, the number of returned source files is often large. It is difficult for developers to learn API usages from a large number of returned results. In order to help developers understand API usages and write API client code more effectively, we have developed an API usage mining framework and its supporting tool called MAPO (for Mining API usages from Open source repositories). Given a query that describes a method, class, or package for an API, MAPO leverages the existing source code search engines to gather relevant source files and conducts data mining. The mining leads to a short list of frequent API usages for developers to inspect. MAPO currently consists of five components: a code search engine, a source code analyzer, a sequence preprocessor, a frequent sequence miner, and a frequent sequence postprocessor. We have examined the effectiveness of MAPO using a set of various queries. The preliminary results show that the framework is practical for providing informative and succinct API usage patterns.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms: Design, Documentation, Measurement.

Keywords: Application Programming Interfaces, Program Comprehension, Mining Software Repositories.

1. INTRODUCTION

During software development, by invoking the corresponding APIs, developers often reuse existing class libraries or frameworks to write client code. These APIs, being equipped with only simple API documents, however, are often complex and not well documented. For example, suppose we plan to use the Byte Code Engineering Library (BCEL) [9] to instrument the bytecode of a Java class by adding an extra method to the class (This programming task was faced by the first author when developing a dynamic analysis tool). By a quick search on BCEL's API document, we can find a class called `org.apache.bcel.generic.ClassGen` containing a method called `public void addMethod(Method m)`, which seems to be the right API method to use. From the API document for this method, we only see a simple description for the method: *"Add a method to this class. Parameters: m - method to add."* We still do not know how to use this method, in particular, how to prepare the `Method` object, what method calls should be invoked on this `Method` object before `addMethod` is invoked, and what method calls are needed to be invoked on the `ClassGen` object before and after the `addMethod` is invoked, and so forth.

Because the API document does not provide sufficient information for us to learn how to use the API, we can search the Web using a general search engine, say Google, to look for other developers' experience of using the API. We can indeed find some articles that include code segments to briefly explain specific usages of BCEL. However, because the same API can be used in different ways, we still do not have high confidence on whether the described code segments represent the API usage that we should follow. We can also use some code search engines such as the Koders search engine [3] and the SPARS-J search engine [5, 13]. These search engines retrieve from open source repositories a long list of source files that contain the call sites of the `addMethod` method. Nevertheless, the numerous and improperly sorted results returned by those source code search engines cannot quickly and comprehensively help us understand the commonality among these source files.

Only collecting a set of call sites or code segments is far from enough to support developers' learning of API usage. Developers are interested in the inherent usage patterns of APIs. Thus, the real challenge is how to construct *a tool to analyze the code segments and disclose the inherent usage patterns*, which motivates this research.

In order to help developers understand API usages and write API client code more effectively, we have developed an API usage mining framework and its supporting tool called MAPO (for Mining API usages from Open source repositories) by leveraging the existing code search engines. The mining produces a short list of frequent API usage patterns for developers to inspect. MAPO consists of five components: a code search engine, a source code analyzer, a sequence preprocessor, a frequent sequence miner, and a frequent sequence postprocessor. To examine its effectiveness, we have applied the MAPO tool on a set of various queries. The preliminary results show that the framework is practical for providing informative and succinct API usage patterns.

MSR'06, May 22–23, 2006, Shanghai, China.

2. MAPO DESIGN CONSIDERATIONS

In MAPO, we want to achieve the following four objectives.

1. The tool should be able to extract API usage information from a source file that may not be able to be compiled by a compiler, because a source code search engine may not return all other source files that the source file depends on.
2. The tool should be able to infer frequent API usages that include sequencing information among method calls. The sequencing information is an important part of API usages. For example, the `open` method of a `File` object needs to be invoked before the `read` method.
3. The tool should be able to mine frequent API usages that include method calls from more than one class, because realistic API usages often involve methods from multiple classes.
4. The tool should be able to produce a short list of relevant frequent API usage patterns for inspection.

3. CHOICES OF MINING TOOLS

Given a set of code segments, a user wants to obtain the common usage patterns of APIs. A few data mining techniques may be applicable in such a situation.

Straightforwardly, for each code segment, we can obtain the set of APIs used in the segment. Then, we can mine the combinations of APIs appearing in many segments by applying the frequent itemset mining methods such as Apriori [6] and FP-growth [10]. Given a transaction database where each transaction is a set of items, and a minimum support threshold min_sup, a frequent itemset mining method returns the complete set of item combinations that appear in at least min_sup transactions.

Frequent itemset mining provides the insights on which APIs are frequently used together in code segments. However, it still does not fully disclose the usage patterns. Particularly, frequent itemsets do not indicate how a group of APIs may be invoked in some specific order.

To capture the groups of APIs that are frequently used together as well as the orders in which they are used, we mine sequential patterns [7]. Given a database of sequences and a minimum support threshold min_sup, a sequential pattern mining algorithm returns the complete set of frequent subsequences, called *sequential patterns*, that appear in at least min_sup sequences in the database.

The complete set of sequential patterns are informative for API usage analysis. It, however, may contain redundant information. For example, suppose methods `open`, `read` and `close` of a `File` object are always called in the order of `open`-`read`-`close`. Then, ⟨`open`⟩, ⟨`read`⟩, ⟨`close`⟩, ⟨`open`; `read`⟩, ⟨`open`; `close` ⟩, ⟨`read`; `close`⟩, and ⟨`open`; `read`; `close`⟩ are all sequential patterns with the same frequency in the database. Pattern ⟨`open`; `read`; `close`⟩ should be used as the representative of the whole group of sequential patterns because it captures the complete usage information that `open`, `read`, and `close` are used. Once pattern ⟨`open`; `read`; `close`⟩ is identified, the other six patterns in the group become redundant because they are sub-patterns of ⟨`open`; `read`; `close`⟩ and have the same frequency. A pattern S such as ⟨`open`; `read`; `close`⟩ is called a *closed sequential pattern* if it is frequent and there exists no any proper super-pattern of S having the same frequency as S. In the API usage mining task, the complete set of closed sequential patterns gives the complete yet non-redundant information on the common usage patterns of APIs.

Finite state automaton (FSA) learning has been frequently used to learn API protocols in the form of an FSA out of program-execution traces [8, 16]. Given a set of sequences, an FSA learning algorithm reconstructs an FSA that can accept these sequences.

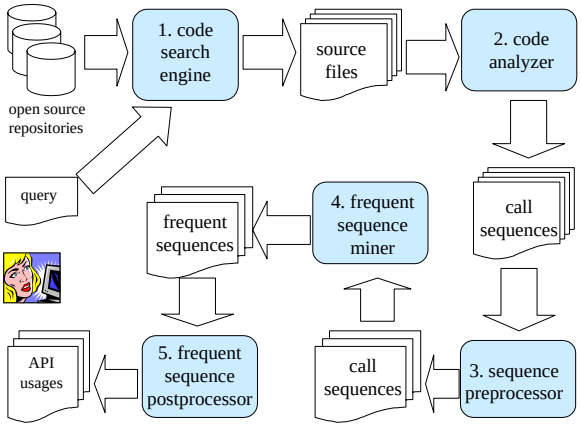


Figure 1: An overview of the API usage mining framework.

In our research context, the sequences extracted from the results of code search engines can include many irrelevant method calls; therefore, applying FSA learning in our research context would produce a large FSA, which is often not useful and not focusing. A probabilistic FSA learning algorithm [18] can be used to infer a probabilistic FSA where each edge is weighted by how often the edge is traversed while accepting sequences. Infrequent behavior reflected by those rarely-traversed edges can be removed to reduce the complexity of the learned FSA. The resulting FSA, however, would still be complicated.

4. API USAGE MINING FRAMEWORK

To automatically mine API usages from open source repositories, we have developed a novel framework based on existing code search engines and a frequent sequence miner. Figure 1 shows the overview of the framework. The framework receives a query describing a method name, class name, or package name, and outputs a set of API usages (in the form of method call sequences). The framework consists of five major components: a code search engine, a code analyzer, a sequence preprocessor, a frequent-sequence miner, and a frequent-sequence postprocessor. The code search engine receives a query and then searches open source repositories for source files that are relevant to the query. The code analyzer analyzes the relevant source files returned by the code search engine and produces a set of method call sequences, each of which is a callee sequence for a method defined in the source files. The sequence preprocessor inlines some call sequences into others based on caller-callee relationships and removes some irrelevant call sequences from the set of call sequences according to the given query. The frequent-sequence miner discovers frequent sequences from the preprocessed sequences. The frequent-sequence postprocessor reduces the set of frequent sequences in some ways. We next illustrate each of the components in the framework in detail.

4.1 Code Search Engine

There exist a number of code search engines[1]. Among the non-academic search engines, we found that Koders [3], CodeBase [1], and DocJar [2] can return a list of Java source files given the textual query of "`bcel`." SPARS-J [5] developed by Inoue et al. [13] is one of the few academic code search engines. Like the preceding non-academic engines, SPARS-J can also return source files given the textual query of "`bcel`." Currently we have not committed our development efforts to develop a tool for automatically grabbing

[1] `http://gonzui.sourceforge.net/links.html`

source files returned by various search engines. Instead, we manually download source files from the returned set of links. We plan to implement a tool to automate this task in the future. Note that although our framework is based on a code search engine, the source files used for API usage mining can also be collected directly from any source repositories, such as local source repositories within a software company or a combination of open and local source repositories.

4.2 Source Code Analyzer

To extract method-call sequences from source files, we have developed a source code analyzer based on a lightweight source code analyzer PMD [4], which does not require source files to be compilable. From each source file, the code analyzer extracts a list of methods, each of which is associated with a sequence of method calls invoked by the method. Currently we count the number of method parameters to characterize method signatures in method-call names. We ignore control flows but simply use call-site locations when extracting method-call sequences. Note that method-call sequences can include method calls from more than one class; therefore, we can later infer from them API usages involving more than one class.

We incorporate various techniques in the source code analyzer to try to collect the class name and the full package name (called full class name) for each method call. The preceding task is not trivial when the classes that a source file depends on are unavailable. We keep track of field declarations and local variable declarations so that we can know the class name of a method call based on the receiver object name. We keep track of `import` statements so that we can know the full package name of a class based on the class name if the class is explicitly exported in the `import` statements. We construct a map from method-call names to their full class names across various source files in the first pass of the analysis. Then, in the second pass, for those method calls whose full class names cannot be found, we assign to them full class names if we can find the same method-call names in the map. In the end, we filter out from the extracted method call sequences those method calls whose full class names cannot be found.

4.3 Sequence Preprocessor

We have developed several techniques to improve the quality of extracted method-call sequences before they are fed to a mining tool. First, we filter out those method calls whose full class names start with "`java.`": those are commonly used Java library classes. Including them in the sequences is often not necessary.

Second, when `ee` is the callee of a caller `er`, and the method-call sequence of `ee` is `ms`, we inline `ee` by replacing all occurrences of `ee` with `ms` in the method-call sequence of `er`. We perform the inlining process for three iterations by default, allowing us to collect method sequences up to the call depth of three. Note that if a call sequence pattern is spread across several source files or the call depth of three, MAPO cannot recognize it completely.

Finally, from the set of inlined sequences, we remove method-call sequences that do not contain the given query entity (e.g., method name, class name, or package name), because these sequences are not relevant to the given query entity.

4.4 Frequent Sequence Miner

We use the BIDE [19] algorithm to mine closed sequential patterns from the preprocessed method-call sequences. BIDE enumerates closed sequential patterns in a depth-first search. For example, suppose A, B, C, and D are the APIs in question. Then, the complete set of closed sequential patterns can be divided into four exclusive subsets: the ones having $\langle A \rangle$, $\langle B \rangle$, $\langle C \rangle$, and $\langle D \rangle$ as prefixes, respectively. Each subset is further divided recursively.

In the depth-first search, once a sequence S is encountered whose frequency in the database is smaller than the support threshold, BIDE does not need to search any longer sequence S' that has S as a prefix, because the frequency of S' cannot exceed that of S.

Moreover, once a frequent sequence S is met, all APIs that appear in every sequence that contains S in the database are also extracted and a closed sequential pattern is formed. On the other hand, if S is a sub-sequence of a closed sequential pattern S' and S and S' have the same frequency, then BIDE does not need to recursively search the subtree of S, because it is not closed.

BIDE also uses a few techniques to speed up the search, such as searching using projected databases and the pseudo-projection technique. Limited by space, we omit the details here.

Table 1: API usage mining results

query	#files	#seqs	#seqs-pre	#freqseq	#freqseq-post
BCEL	36	1087	186	429	8
Javassist	50	828	141	90	23

4.5 Frequent Sequence Postprocessor

Because the number of frequent sequences mined by BIDE could be large, we have developed several techniques to reduce the size of frequent sequences without compromising important API usage information. First, we remove frequent sequences that do not contain the given query entity (e.g., method name, class name, or package name), because these frequent sequences are not relevant to the query entity. Second, in frequent sequences, we compress consecutive calls of the same method into one. Alteratively we can compress method-call sequences in a similar way in the sequence preprocessor but doing compression here can help compress repetitive call patterns separated by infrequent method calls in original call sequences. Third, we remove duplicate frequent sequences after the compression. Finally, we further reduce the set of frequent sequences so that every frequent sequence in the reduced set is not a subsequence of another in the reduced set. We adapted the implementation of the longest common sequences (LCS) [11] algorithm to implement the subsequence checking.

5. PRELIMINARY RESULTS

We have applied MAPO on various queries. This section shows two particular queries that are related to the motivating example shown in Section 1. We searched Koders [3] with two queries: the textual BCEL query of "`org.apache.bcel.generic ClassGen addMethod`" and the textual Javaassist query of "`javassist CtClass addMethod`," where we replace those dots that separate package names, class names, and method names with space characters in order to allow Koders to return more related results. The method in the Javaassist query has the same functionality as the method in the BCEL query but these two methods are from two different libraries.

Table 1 show the statistics of the API usage mining results for these two queries. Columns 1-6 show the query name, the number of source files returned by Koders, the number of sequences extracted by the code analyzer, the number of sequences produced by the sequence preprocessor, the number of frequent sequences produced by BIDE, and the number of frequent sequences produced by the frequent sequence postprocessor, respectively. From the statistics, we can observe that the sequence preprocessor is effective in reducing the size of sequences before being fed to BIDE, and the frequent sequence postprocessor is also effective in reducing the size of frequent sequences before being inspected by users.

We inspected the frequent patterns produced by MAPO and found them precise in general in characterizing the API usages. For example, for the BCEL query, the first frequent sequence is listed as below where package names are omitted for simplicity, the enclosed numbers represent the total number of method parameters if any, and "<init>" represents a constructor call.

```
InstructionList.<init>
InstructionFactory,createLoad(2)
InstructionList,append(1)
InstructionFactory,createReturn(1)
InstructionList,append(1)
MethodGen,setMaxStack
MethodGen,setMaxLocals
MethodGen,getMethod
ClassGen,addMethod(1)
InstructionList.dispose
```

The API usages mined by MAPO for the Javassist query are simpler and also more diverse than the ones for the BCEL query. The first frequent sequence is simply two method calls:

```
CtNewMethod,make(2)
CtClass,addMethod(1)
```

6. RELATED WORK

CodeWeb developed by Michail [17] mines association rules such as that application classes inheriting from a particular library class often instantiate another class or one of its descendants. MAPO focuses on API usages in general beyond library reuse patterns through class inheritances. In addition, MAPO mines API usages that include sequencing information among method calls. PR-Miner developed by Li and Zhou [14] uses frequent itemset mining to extract implicit programming rules and detect their violations for detecting bugs. The rules mined by PR-Miner do not include sequencing information, which is mined by MAPO. A tool developed by Williams and Hollingsworth [20] and DynaMine developed by Livshits and Zimmermann [15] mine simple rules from software revision histories. These rules involve mostly method pairs, whereas MAPO mines more complicated API usage patterns from code segments returned by a code search engine. Different from all the preceding mining tools, which take no query before mining, MAPO takes a query and mines from code segments relevant to the query.

MAPO is related to a number of code search engines [3, 5, 13] as well as the Strathcona tool developed by Holmes and Murphy [12]. Strathcona locates a set of relevant code examples from an example repository by matching the structure of the code under development with the code examples in the repository. Like other code search engines, Strathcona returns a list of relevant code examples, whereas MAPO can extract common patterns among the list of relevant code examples returned by a code search engine or even Strathcona.

7. CONCLUSIONS

In order to help developers understand API usages and write API client code more effectively, we have developed a novel framework and its supporting tool called MAPO for mining API usages from open source repositories by leveraging existing code search engines and a frequent sequence miner. MAPO can produce a short list of succinct frequent sequences for inspection. Our preliminary results show that MAPO provides useful API usage patterns for developers to inspect. In future work, we plan to synthesize code fragments from mined frequent API usages. These synthesized code fragments can be directly inserted into developers' code.

Acknowledgments

We would like to thank Jiawei Han and Jianyong Wang for helping us to use the BIDE tool.

8. REFERENCES

[1] CodeBase, 2005. http://www.codase.com/.
[2] DocJar, 2005. http://www.docjar.com/.
[3] The Koders source code search engine, 2005. http://www.koders.com.
[4] PMD, 2005. http://pmd.sourceforge.net/.
[5] SPARS-J, 2005. http://demo.spars.info/.
[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Sept. 1994.
[7] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering*, pages 3–14, Taipei, Taiwan, Mar. 1995.
[8] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.
[9] M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. http://jakarta.apache.org/bcel/.
[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data*, pages 1–12, May 2000.
[11] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24:644–675, 1977.
[12] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. 27th International Conference on Software Engineering*, pages 117–125, 2005.
[13] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, March 2005.
[14] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*, pages 306–315, 2005.
[15] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proc. ESEC/FSE*, pages 296–305, 2005.
[16] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *Proc. 10th International Conference on Engineering of Complex Computer Systems*, pages 292–301, June 2005.
[17] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. 22nd International Conference on Software Engineering*, pages 167–176, 2000.
[18] A. V. Raman and J. D. Patrick. The sk-strings method for inferring pfsa. In *Proc. Workshop on Automata Induction, Grammatical Inference and Language Acquisition*, 1997.
[19] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. 20th International Conference on Data Engineering*, pages 79–90, 2004.
[20] C. C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proc. 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

Program Element Matching for Multi-Version Program Analyses

Miryung Kim, David Notkin
Computer Science & Engineering
University of Washington
Seattle, WA
{miryung,notkin}@cs.washington.edu

ABSTRACT

Multi-version program analyses require that elements of one version of a program be mapped to the elements of other versions of that program. Matching program elements between two versions of a program is a fundamental building block for multi-version program analyses and other software evolution research such as profile propagation, regression testing, and software version merging.

In this paper, we survey matching techniques that can be used for multi-version program analyses and evaluate them based on hypothetical change scenarios. This paper also lists challenges of the matching problem, identifies open problems, and proposes future directions.

Categories and Subject Descriptor: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement —*restructuring, reverse engineering, and reengineering*

General Terms: Documentation, Algorithms

Keywords: matching, software evolution, multi-version analysis

1. INTRODUCTION

In the last several years, researchers in software engineering have begun to analyze programs together with their change history. In contrast to traditional program analyses that examine a single version, multi-version program analyses use multiple versions of a program as input and mine change patterns.

There are roughly two different types of multi-version analyses: (1) coarse-grained analyses and (2) fine-grained analyses. Coarse-grained analyses compute changes between two consecutive versions of a program, aggregate the change information across multiple versions or across multiple files, and infer coarse-grained patterns [37, 15, 20, 17]. For example, Nagappan and Ball's analysis [37] finds line-level changes between two consecutive versions, counts the total number of changes per binary module, and infers the characteristics of frequently changed modules. On the other hand, fine-grained analyses track how individual code fragments changed during program evolution to infer fine-grained change patterns [29, 31, 52, 38, 43, 48, 51, 11]. For example, a clone genealogy extractor tracks individual code snippets over multiple versions to infer clone evolution patterns [29]. As another example, a signature change pattern analysis [31, 30] traces how the name and the signature of functions change. Matching elements between two versions of a program is a fundamental building block for fine-grained multi-version analyses as well as other software evolution research such as software version merging, regression testing, profile propagation, etc [36, 42, 21, 46].

We first define the problem of matching code elements between two versions:

Suppose that a program P' is created by modifying P. Determine the difference Δ between P and P'. For a code fragment $c' \in P'$, determine whether $c' \in \Delta$. If not, find c''s corresponding origin c in P.

The problem definition states that we must compute the difference between two programs. Computing semantic differences requires solving the problem of semantic program equivalence, which is an undecidable problem. Thus, once the problem is approximated by matching a code element by its syntactic and textual similarity, solutions depend on the choices of (1) an underlying program representation, (2) matching granularity, (3) matching multiplicity, and (4) matching heuristics. In this paper, we explain how the choices impact applicability of each matching method and how the choices affect effectiveness and accuracy of matching by creating an evaluation framework for existing matching techniques.

The rest of this paper is organized as follows. The next section discusses several multi-version analyses, which demonstrate the needs of program element matching. Then, we discuss challenges of program element matching in Section 3. Section 4 presents a survey of state-of-the-art matching techniques from various research areas such as multi-version program analyses, profile propagation, software version merging, and regression testing. Section 5 compares the surveyed techniques and Section 6 evaluates each technique using hypothetical program change scenarios. Section 7 presents open problems and future directions.

2. MOTIVATING PROBLEMS

In this section, we describe several multi-version analysis problems that demonstrate importance of program element matching.

The first problem is maintaining two similar programs

MSR'06, May 22–23, 2006, Shanghai, China.

that originated from the same source but evolved differently in parallel.[1] In many organizations, it is a common practice to clone a product or a module and maintain the clones in parallel [13]. Maintenance difficulties arise when programmers discover a critical bug in cloned parts. If programmers do not know whether the discovered bug is relevant to other cloned counterparts, they must inspect source code of the counterparts. We believe that programmers can better locate relevant counterparts by understanding how clones change over time. Monitoring clones requires tracking each clone by its physical location such as a file name, a procedure name, or calibrated line numbers [29].

Our second motivating problem is understanding the evolution of information hiding interfaces. The information hiding principle [41] states that programmers should anticipate what kinds of decisions are likely to change in the future and hide them using an interface. In general, it is difficult for programmers to predict which design decisions are likely to change; thus, unanticipated design decisions result in degradation of original software design. We believe that understanding interface evolution can shed light on (1) which decisions were originally hidden but later exposed and (2) how unanticipated decisions impact original interface design.

In addition to the problems above, type change analysis [38], signature change analysis [31], and instability analysis [11] also require matching program elements in order to track code elements over time.

3. MATCHING CHALLENGES

This section lists challenges of program element matching.

3.1 Absence of Benchmarks

It is difficult to evaluate matching techniques because there is no reference data set or archive of editing logs. Previous studies [30] also indicate that programmers often disagree about the origin of code snippets; low inter-rater agreement suggests that there may be no ground truth in program element matching.

3.2 Various Granularity Support

With respect to our motivating problems in Section 2, we cannot assume that programmers intend to track program elements at a fixed granularity. There are two reasons why matching techniques must support various granularity. First, a programmer may want to track design decisions that cannot be mapped to program units precisely. Second, a programmer may want track program elements at a different granularity depending on the nature of tasks. For example, if matching techniques are to be used for profile propagation or precise regression test selection, mappings should be found at a fine granularity such as at the level of control flow graph edges [42] or at the level of code blocks [46, 44]. On the other hand, if a programmer wants to use matching results for program understanding tasks, it would be more appropriate to find associations at a higher level such as a file.

3.3 Types of Code Changes

Certain types of code changes make the matching problem non-trivial. For example, tracking code by its enclosing procedure name would fail if programmers merged, split, or renamed procedures. When a programmer copies code, matching techniques cannot assume one-to-one mappings between old elements and new elements because an old code element can have more than one matching descendants in a new version.

[1]In an open source project community, this practice is often called as forking.

4. MATCHING TECHNIQUES

This section describes matching techniques used for software version merging, program differencing, profile propagation, regression testing, and multi-version program analyses. For easy comparison, we group techniques by program representation. We describe clone detectors and tools that infer refactoring events in Section 4.7 and 4.8 because these tools can be leveraged for finding correspondences between two versions.

4.1 Entity Name Matching

The simplest matching method treats program elements as immutable entities with a fixed name and matches the elements by name. For example, Zimmermann et al. modeled a function as a tuple, *(file name, FUNCTION, function name)*, and a field as a tuple, *(function name, FIELD, field name)* [51]. Similarly, Ying et al. [48] modeled a file with its full path name. In fact, matching by name would be sufficient for many multi-version analyses that intend to identify coarse-grained patterns such as the characteristics of fault prone modules [15, 20, 37].

4.2 String Matching

When a program is represented as a string, the best match between two strings is computed by finding the longest common subsequence (LCS) [7]. The LCS problem is built on the assumption that (1) available operations are addition and deletion, and (2) matched pairs cannot cross one another. Thus, the longest common subsequence does not necessarily include all possible matches when available edit operations include copy, paste, and move. Tichy [45] (*bdiff*) extended the LCS problem by relaxing the two assumptions above: permitting crossing block moves and not requiring one-to-one correspondence.

The line-level LCS implementation, *diff* [25], has served as basis for many multi-version analyses, because (1) *diff* is fast and reliable, and (2) popular version control systems such as CVS [2] or Subversion [1] already store changes as line-level differences. For example, a clone genealogy extractor tracks code snippets by their file name and line number [29]. As another example, fix-inducing code snippets [43] are inferred by tracking backward a tuple of *(file name:: function name:: line number)* from the moment that a bug is fixed.

4.3 Syntax Tree Matching

For software version merging, Yang [47] developed an AST differencing algorithm. Given a pair of two functions (f_T, f_R), the algorithm creates two abstract syntax trees T and R and attempts to match the two tree roots. If the two roots match, the algorithm aligns T's subtrees $t_1, t_2, ..., t_i$ and R's subtrees $r_1, r_2, ...r_j$ using the LCS algorithm and matches subtrees recursively. This type of tree matching respects the parent-child relationship as well as the order between sibling nodes, but is very sensitive to changes in nested block and control structures because tree roots must be matched for every level.

Hunt and Tichy do not directly compare ASTs but use syntactic information to guide string level differencing. Their 3-way merging tool [24] parses a program into a language neutral form, compares token strings using the LCS algorithm, and finds syntactic changes using structural information from the parse.

For dynamic software updating, Neamtiu et al. [38] built an algorithm that tracks simple changes to variables, types, and functions based on a AST representation. Neamtiu's algorithm assumes that function names are relatively stable over time and matches the ASTs of functions with the same name; the algorithm traverses two ASTs in parallel and incrementally adds one-to-one mappings as long as the ASTs have the same shape. In contrast to Yang's algorithm, Neamtiu's algorithm cannot compare structurally different ASTs.

4.4 Control Flow Graph Matching

Laski and Szermer [33] first developed an algorithm that computes one-to-one correspondences between CFG nodes in two programs $P1$ and $P2$. This algorithm first reduces a CFG to a series of single-entry, single-exit subgraphs called hammocks and matches a sequence of hammock nodes using a depth first search (DFS). Once a pair of corresponding hammock nodes is found, the hammock nodes are recursively expanded in order to find correspondences within the matched hammocks.

Jdiff [5] extends Laski and Szermer's (LS) algorithm to compare Java programs based on an enhanced control flow graph (ECFG). *Jdiff* is similar to the LS algorithm in the sense that hammocks are recursively expanded and compared, but is different in three ways: First, while the LS algorithm compares hammock nodes by the name of a start node in the hammock, *Jdiff* checks whether the ratio of unchanged-matched pairs in the hammock is greater than a chosen threshold in order to allow for flexible matches. Second, while the LS algorithm uses DFS to match hammock nodes, *Jdiff* only uses DFS up to a certain look-ahead depth to improve its performance. Third, while the LS algorithm requires hammock node matches at the same nested level, *Jdiff* can match hammock nodes at a different nested level; thus, *Jdiff* is more robust to addition of while loops or if-statements at the beginning of a code segment. *Jdiff* has been used for regression test selection [40] and dynamic impact analysis [6].

4.5 Program Dependence Graph Matching

There are several program differencing algorithms based on a program dependence graph [23, 12, 26]. These algorithms are not applicable to popular modern program languages because they can run only on a limited subset of C languages without global variables, pointers, arrays, or procedures.

4.6 Binary Code Matching

BMAT [46] is a fast and effective tool that matches two versions of a binary program without knowledge of source code changes. *BMAT* was used for profile propagation and regression test prioritization [44]. BMAT's algorithm matches blocks in three steps. The first step of BMAT's matching algorithm is to find one-to-one mappings between the procedures in two versions based on their names, type information, and code contents. To match procedures with different names, block trial matching is done on procedure pairs with a small number of character differences in their hierarchical names. In this step, the thresholds for procedure name difference and block matching percentage are both set heuristically. In the second step, BMAT first matches data blocks within each pair of matched procedures using a hash function and matches remaining unmatched data blocks if the unmatched blocks are sandwiched by already matched pairs. In the third step, BMAT matches code blocks in multiple hashing passes. During hash-based matching, if hash values collide, two heuristics are used to break ties: (1) crossing matches are forbidden at certain hashing passes, and (2) a pair of blocks is preferred to other tied pairs if either its predecessors or successors are also matched. For remaining unmatched blocks, BMAT matches blocks based on control flow equivalence, allowing one-to-many mappings between old code blocks and new code blocks.

4.7 Clone Detection

A clone detector is simply an implementation of an arbitrary equivalence function. The equivalence function defined by each clone detector depends a program representation and a comparison algorithm. Most clone detectors [8, 28, 9, 32, 27] are heavily dependent on (1) hash functions to improve performance, (2) parameterization to allow flexible matches, and (3) thresholds to remove spurious matches. A clone detector can be considered as a many-to-many matcher based solely on content similarity heuristics.

4.8 Origin Analysis Tools

Origin analysis infers refactoring events such as splitting, merging, renaming and moving by comparing two versions of a program [14, 52, 31, 4, 18, 35, 19]. Origin analysis tackles the program element matching problem directly but produces matching results only at a predefined granularity such as a procedure, class or file.

Demeyer et al. [14] first proposed the idea of inferring refactoring events by comparing the two programs. Demeyer et al. used a set of ten characteristics metrics for each class, such as LOC and the number of method calls within a method (i.e., fan-out) and inferred where refactoring events occurred based on the metric values and a class inheritance hierarchy.

Zou and Godfrey's origin analysis [52] matches procedures using multiple criteria (names, signatures, metric values, and a set of callers and callees) and infers merging, splitting, and renaming events. Both Demeyer et al. and Zou and Godfrey's analyses are semi-automatic in the sense that a programmer must manually tune matching criteria and select a match among candidate matches.

Kim et al. [30] automated Zou and Godfrey's procedure renaming analysis. In addition to matching criteria used by Zou and Godfrey, Kim et al. used clone detectors such as CCFinder [28] and Moss [3] to calculate content similarity between procedures. An overall similarity is computed as a weighted sum of each similarity metric, and a match is selected if the overall similarity is greater than a certain threshold. To create an evaluation data set, ten human judges identified renaming events in the Subversion and the Apache projects, and if seven out of ten judges agreed the origin of a renamed procedure, a match was added to a reference data set. Using the reference data set, Kim et al. optimized each factor's weight and tuned the threshold

Table 1: Comparison of Matching Techniques

Program Representation	Citation	Granularity	Assumed Correspondence	Multiplicity	Heuristics N	Heuristics P	Heuristics S	Application
A set of entities	[20, 15, 37]	Module		1:1	√			Fault proneness
	Bevan et al. [11]	File		1:1	√			Instability
	Ying et al. [48]	File		1:1	√			Co-change
	Zimmermann et al. [51]	File Procedure Field		1:1	√			
String	*diff* [25]	Line	File	1:1			√	Merging Clone genealogy [29] Fix inducing code [43]
	bdiff [45]	Line	File	1:n			√	Merging
AST	*cdiff* [47]	AST Node	Procedure	1:1	√			
	Neamtiu et al.[38]	Type, Var		1:1	√	√		Type change
	Hunt, Tichy[24, 35]	Token	File	1:1		√	√	Merging
CFG	*Jdiff* [5]	CFG node		1:1	√		√	Regression testing Impact analysis
Binary	BMAT [46]	Code block		1:1 (procedure) n:1 (block)	√	√	√	Profile propagation Regression testing
Hybrid	Zou, Godfrey [52]	Procedure		1:1 or 1:n or n:1	√		√	Origin analysis
	Kim et al. [30]	Procedure		1:1	√		√	Signature change [31] Renaming analysis

N: Name-based heuristics, P: Position-based heuristics, S: Similarity-based heuristics

value. The accuracy of Kim's tool was better than the average accuracy of human judges, indicating that human judges significantly disagreed about the origin of procedures.

5. COMPARISON

Table 1 shows comparison of the state-of-the-art matching techniques in Section 4. As shown in the fourth column of Table 1, many matching techniques assume correspondence at a certain granularity no matter whether this assumption is explicitly stated or not. For example, using *diff* to match code snippets assumes that input files already are matched. As another example, using *cdiff* to match AST nodes assumes that enclosing functions are matched by the same name.

All matching techniques heavily rely on heuristics to reduce a matching scope and to improve precision and recall. The heuristics are categorized into three categories: [2]

1. Name-based heuristics match entities with similar names. For example, BMAT and Jdiff match procedures in multiple phases by the same globally qualified name (e.g., System.out.println), by the same hierarchical name, by the same signature, and by the same name.
2. Position-based heuristics match entities with similar positions. If entities are placed in the same syntactic position or surrounded by already matched pairs (i.e., a sandwich heuristic), they become a matched pair. For example, BMAT uses a sandwich heuristic aggressively to remove unmatched pairs. As another example, Neamtiu's algorithm traverses two ASTs in parallel and matches variables placed in the same syntactic position regardless of their labels.
3. Similarity-based heuristics match entities that are nearly identical; they often rely on parameterization and a hash function to find near identical entities. All clone detectors can be viewed as a similarity-based matcher.

[2]The three categories are not comprehensive or mutually exclusive.

The three different types of heuristics complement one another. For example, when hash values collide or parameterization results in spurious matches, position-based heuristics will select a matched pair that preserves linear ordering or structural ordering by checking neighboring matches. Table 1 (column 6 to 8) summarizes which kinds of heuristics that each matching technique uses.

6. EVALUATION

Matching techniques are often inadequately evaluated—Only Kim et al. conducted a comparative study using human subjects [30]. This lack of evaluation is exacerbated by the fact that there are no agreed evaluation criteria or representative benchmarks. Finding such universal criteria would be difficult since each technique is built for a different goal. For example, matching techniques for regression testing or profile propagation [5, 46, 49] can be evaluated by the accuracy of static branch prediction and code coverage; but even this evaluation method is not applicable to programs without test suites. To evaluate matching techniques uniformly, we take a scenario-based evaluation approach; we design a small set of hypothetical program change scenarios, on which we describe how well various matching techniques will perform. [3]

Scenario 1: (1) a programmer inserts if-else statements in the beginning of the method m_A, and (2) the programmer reorders several statements in the method m_B without changing semantics as shown in Table 2.

The ideal matching technique should produce (s1-s1'), (s2-s2'), (s3-s4'), (s4-s3'), and (s5-s5') and identify that s0' is added. The third column of Table 3 summarizes how well each technique will work in this scenario. *Diff* can match lines of m_A but cannot match reordered lines in m_B because the LCS algorithm does not allow crossing block moves. On the other hand, *bdiff* can match reordered lines in m_B because crossing block moves are allowed. Neamtiu's algo-

[3]PDG-based matching techniques are excluded due to lack of modern programming language support.

Table 2: Scenario 1 Code Change

before	after

`mA (){`
`  if (pred_a) { \\s1`
`  foo() \\s2`
`  }`
`}`
`mB (b){`
`  a:= 1 \\s3`
`  b:= b+1 \\s4`
`  fun(a,b) \\s5`
`}`
	`mA (){`
	`  if (pred_a0) { \\s0'`
	`  if (pred_a) { \\s1'`
	`  foo() \\s2'`
	`  }`
	`  }`
	`mB (b){`
	`  b:= b+1 \\s3'`
	`  a:= 1 \\s4'`
	`  fun(a,b) \\s5'`
	`}`

rithm will perform poorly in both m_A and m_B because it does not perform a deep structural match. *Cdiff* cannot match unchanged parts in m_A correctly because *cdiff* stops early if roots do not match for each level. *Jdiff* will be able to skip the changed control structure, map unchanged parts in m_A, and match reordered statements in m_B if the look-ahead threshold is greater than the depth of nested controls. *BMAT* cannot track code blocks in m_B because BMAT's hashing algorithms are instruction order sensitive. In conclusion, *Jdiff* will work the best for changes within procedures at a statement or predicate level.

Scenario 2: A file PElmtMatch changed its name to PMatching. A procedure matchBlck are split into two procedures matchDBlck and matchCBlck. A procedure matchAST changed its name to matchAbstractSyntaxTree.

The ideal matching technique should produce (PElmtMatch, PMatching), (matchBlck, matchDBlck), (matchBlck, matchCBlck), and (matchAST, matchAbstractSyntaxTree). The fourth column of Table 3 summarizes how each technique will work in this scenario. Most name-based matching techniques will do poorly due to renaming events. *Diff* and *bdiff* will be able to track each line only if file names did not change. Both *cdiff* and Neamtiu's algorithm will perform poorly if procedure names changed. Both *BMAT* and origin analysis tools will perform well because they rely on multiple passes of hash functions and multiple phases of name matching.

The remaining columns of Table 3 describe how well each matching technique will work in case of restructuring tasks at a procedure level or at a file level.

Based on Table 1 and 3, we conclude the following:

- Matching techniques based on AST or CFG produce matches at fine-grained levels but are only applicable to a complete and parsable program. Researchers must consider the trade-off between matching granularity, matching requirements, and matching cost.
- Many techniques employ the LCS algorithm even when matching AST or CFG, thus inheriting the assumptions of LCS: one-to-one correspondences between matched entities and linear ordering among matched pairs. This sort of implicit assumptions must be carefully examined before implementing a matcher.
- Most techniques support only one-to-one mappings at a fixed granularity. Therefore, they will perform poorly when merging or splitting occurs.
- The more heuristics are used, the more matches can be found by complementing one another. For example, name-based matching is easy to implement and can reduce matching scope quickly, but it is not robust to renaming events. In this case, similarity-based matching can produce matches between renamed entities and position-based matching can leverage already matched pairs to infer more matches.

7. FUTURE DIRECTIONS

This section lists remaining open problems and future directions.

Hybrid Matcher. Although no single technique performs perfectly in all change scenarios but the combination of all techniques does. Thus combining multiple techniques may improve the accuracy of matches by complementing one another. The simplest way is to run all matching techniques separately and find consensus among the results. Another way of building a hybrid matcher is to leverage a feedback loop between matching tools and tools that infer refactoring events. Determining which refactoring occurred and determining correspondences is a chicken and egg problem; inferring refactoring events requires knowledge of correspondences, and finding good correspondences is achieved by knowing which refactoring occurred. This feedback cycle provides an opportunity to find more matches. The results of inferred transformations are fed to matching tools, and the matching results are fed back to a refactoring reconstruction tool iteratively until optimal correspondences are found.

We must note that combining results from multiple matchers will require tremendous efforts because (1) not every matching tool is available for public use or applicable to popular programming languages and (2) different matchers use different program representations.

Capturing Editing Operations. Having a complete history of logical editing operations would nullify the matching problem. However, most software repositories employ state-based merging not operation-based merging [34], thus making it impossible to restore logical editing operations completely. Even when an edit log is available, if editing operations are captured at a key stroke level, it is not trivial to reconstruct logical editing operations (such as procedure renaming, splitting, and merging) and produce matches between program elements. Recently, capturing and replaying refactoring operations is shown to be possible in an Eclipse IDE [22], so we can leverage this type of refactoring history to initiate the feedback loop discussed above.

Interval Manipulation vs. Matching Tool Selection. In this paper, we simplified a multi-version program matching problem as a two version matching problem. To use a matching technique in the context of multi-version analyses, the interval between each pair of versions must be determined. In the past, the granularity of available historical data limited a sampling interval for multi-version analyses. Recently, several infrastructures [10, 51, 50] were built to facilitate multi-version analyses by restoring commit transactions from a source code repository and automatically extracting multiple versions separated by an arbitrary time interval. These infrastructures enable multi-version analyses to manipulate a sampling interval. Therefore, the remaining problem is to determine an optimal sampling interval for each matching technique (or select an appropriate

Table 3: Evaluation of the Surveyed Matching Techniques

Program Representation	Citation	Scenario		Transformations				Strength and Weakness
				Split/Merge		Rename		
		1	2	Proc	File	Proc	File	
String	*diff* [25]	☒	□	□	□	☒	□	− sensitive to file name changes
	bdiff [45]	■	□	☒	□	☒	□	+ can trace copied blocks
AST	*cdiff* [47]	□	□	□	□	□	□	− sensitive to nested level change − require procedure level mappings
	Neamtiu et al. [38]	□	□	□	□	□	□	− partial AST matching
	Hunt, Tichy [24, 35]	☒	□	□	□	■	□	− require file level mappings + can identify procedure renaming
CFG	*JDiff* [5]	■	☒	□	□	☒	☒	+ robust to signature changes − sensitive to control structure changes
Binary	BMAT [46]	□	■	□	□	■	■	+ robust to procedure renaming − assume 1:1 procedure correspondence − only applicable to binary programs
Hybrid	Zou, Godfrey [52]	□	■	■	■	■	■	− semi-automatic, manual analysis
	Kim et al. [30]	□	■	□	□	■	■	− assume 1:1 procedure correspondence

■ good ☒ mediocre □ poor

matching tool depending on the logical gap between two versions of a program). Another interesting open question is, "can we design a matching technique that works as well as aggregating results from a set of program snapshots that separated by only small changes?"

Matching Result Aggregation. As matching complexity increases by supporting multiple granularities and many-to-many mappings, representing match results becomes a non-trivial problem. In addition, when a two-version matching tool is used for multi-version program analyses, aggregating individual matching results and representing the final results in a compact form remains as an open problem.

Leveraging Dynamic Information. Most matching techniques are based on syntactic similarities at a source code level. In comparison checking research [49, 39], dynamic information has been used to match an optimized version and an unoptimized version of the same program when the two versions were executed on the same input. Abstraction of multiple execution traces may guide matching of a static program representation. For example, comparing dynamic invariants [16] may be useful for identifying variable level matches at the entry (or exit) of a function.

8. CONCLUSION

In this paper, we defined the program element matching problem and argued its importance for fine-grained multi-version analyses. We presented a survey of matching techniques from various research areas and evaluated them based on hypothetical program change scenarios by hand. We believe that our assessment of existing techniques will guide researchers to choose an appropriate matching technique for their analysis.

In conclusion, every matching technique is an implementation of some pseudo equivalence function, and the more heuristics are used, the better the matching technique will work. One direction of future work involves building a hybrid matcher that leverages a feedback loop between matching tools and tools that infer refactoring events. Another future direction is to customize existing matchers in the context of a specific type of multi-version analysis and build an evaluation data set for that analysis. In addition, determining an optimal sampling interval for each matching technique remains as an open problem.

Our longer-term objectives are to (1) define the problem more precisely, allowing for better assessment and sharing of the approaches and (2) lay a foundation for more effective solutions applicable to specific kinds of multi-version analyses.

9. ACKNOWLEDGMENTS

We thank Dagstuhl 05261 seminar participants for fruitful discussions. We also thank Michael Toomim for reading our draft and Vibha Sazawal, Dan Grossman, and Rob DeLine for discussions that helped us refine our ideas.

10. REFERENCES

[1] subversion.tigris.org.
[2] www.cvshome.org.
[3] A. Aiken. A system for detecting software plagiarism.
[4] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE*, pages 31–40, 2004.
[5] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE*, pages 2–13. IEEE Computer Society, 2004.
[6] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE*, pages 432–441, 2005.
[7] A. Apostolico and Z. Galil, editors. *Pattern matching algorithms*. Oxford University Press, UK, 1997.
[8] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.
[9] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.
[10] J. Bevan, J. E. James Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with Kenyon. In *ESEC/FSE*, pages 177–186, 2005.
[11] J. Bevan and E. J. W. Jr. Identification of software instabilities. In *WCRE*, pages 134–145, 2003.

[12] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM TOSEM*, 4(1):3–35, 1995.

[13] J. R. Cordy. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *IWPC '03*, page 196, 2003.

[14] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00*, pages 166–177, 2000.

[15] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[16] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D. Disseratation, University of Washington, Seattle, Washington, Aug. 2000.

[17] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, pages 190–197, 1998.

[18] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05*, pages 29–35.

[19] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *IWPC*, pages 205–214, 2005.

[20] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[21] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA '01*, pages 312–326, 2001.

[22] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05*, pages 274–283, 2005.

[23] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI'90*, volume 25, pages 234–245, June 1990.

[24] J. J. Hunt and W. F. Tichy. Extensible language-aware merging. In *ICSM*, pages 511–520, 2002.

[25] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[26] D. Jackson and D. A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *ICSM '94*, pages 243–252, 1994.

[27] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183. IBM Press, 1993.

[28] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[29] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE*, pages 187–196, 2005.

[30] S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE*, 2005.

[31] S. Kim, E. J. Whitehead, and J. Bevan. Analysis of signature change patterns. In *MSR '05*, pages 64–68.

[32] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[33] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *ICSM*, 1992.

[34] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE'92*, pages 78–87, 1992.

[35] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *Autom. Softw. Eng.*, 10(2):183–202, 2000.

[36] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[37] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE*, pages 284–292, 2005.

[38] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR'05*, pages 2–6.

[39] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI '00*, pages 83–94, 2000.

[40] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12*, pages 241–251, 2004.

[41] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[42] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, 1997.

[43] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05*, pages 24–28, 2005.

[44] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02*, pages 97–106, 2002.

[45] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2(4):309–321, 1984.

[46] Z. Wang, K. Pierce, and S. McFarling. BMAT - a binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism*, 2, 2000.

[47] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[48] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[49] X. Zhang and R. Gupta. Matching execution histories of program versions. In *ESEC/SIGSOFT FSE*, pages 197–206, 2005.

[50] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR'04*, pages 2–6.

[51] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.

[52] L. Zou and M. W. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.

Detecting Similar Java Classes Using Tree Algorithms

Tobias Sager, Abraham Bernstein, Martin Pinzger, Christoph Kiefer
Department of Informatics
University of Zurich, Switzerland
tsager@gmx.ch {bernstein,pinzger,kiefer}@ifi.unizh.ch

ABSTRACT

Similarity analysis of source code is helpful during development to provide, for instance, better support for code reuse. Consider a development environment that analyzes code while typing and that suggests similar code examples or existing implementations from a source code repository. Mining software repositories by means of similarity measures enables and enforces reusing existing code and reduces the developing effort needed by creating a shared knowledge base of code fragments. In information retrieval similarity measures are often used to find documents similar to a given query document. This paper extends this idea to source code repositories. It introduces our approach to detect similar Java classes in software projects using tree similarity algorithms. We show how our approach allows to find similar Java classes based on an evaluation of three tree-based similarity measures in the context of five user-defined test cases as well as a preliminary software evolution analysis of a medium-sized Java project. Initial results of our technique indicate that it (1) is indeed useful to identify similar Java classes, (2) successfully identifies the ex ante and ex post versions of refactored classes, and (3) provides some interesting insights into within-version and between-version dependencies of classes within a Java project.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; E.1 [**Data Structures**]: Trees; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*retrieval models*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Tree Similarity Measures, Software Repositories, Change Analysis, Software Evolution

MSR'06, May 22–23, 2006, Shanghai, China.

1. INTRODUCTION

Similarity analysis of source code is helpful during development to provide, for instance, better support for code reuse, faster prototyping, and clone detection. Consider a development environment that analyzes code while typing and that suggests similar code examples or existing implementations from a source code repository. Mining software repositories by means of similarity measures enables, for instance, code reuse and reduces the development effort (and thus cost) by making the shared knowledge base of code fragments in the repository better accessible. As another software evolution-based scenario, consider software project analysis: the detection of similar entities (Java classes in our case) in a complete project can indicate a deficit in the architecture or implementation flaws. Removing or merging similar classes may increase the overall quality as well as maintainability of a software project.

The goal of this paper is to present an approach to detect similar Java classes based upon their abstract syntax tree (AST) representations. These trees are generated using Eclipse's [10] JDT API in which all statements and operations of Java source code are represented. The generated complete ASTs are converted into an intermediary model called FAMIX (FAMOOS Information Exchange Model) [13, 14]. FAMIX is a programming language-independent model for representing object-oriented source code. The similarity between two classes is computed by tree comparison algorithms comparing the FAMIX tree representations of the two classes. We implemented this process as an Eclipse plug-in called *Coogle* (short for Code Google™). Our initial results show that Coogle is indeed useful to find similar Java classes within a Java software project.

The rest of this paper is structured as follows: next, we introduce our current implementation including the preprocessing in Eclipse and the three implemented tree comparison algorithms: bottom-up maximum common subtree isomorphism, top-down maximum common subtree isomorphism, and the tree edit distance. We then evaluate the effectiveness of these algorithms in the context of five constructed test-cases and a real-world Java project (Section 3), which leads to a discussion of our technique in Section 4. Section 5 reflects on our approach in the light of related work. Finally, we close with our conclusions in Section 6.

2. OUR APPROACH: COOGLE

We implemented a first prototype as an Eclipse plug-in [10] called *Coogle* that stands for Code Google™. Coogle essentially implements the following two steps to determine

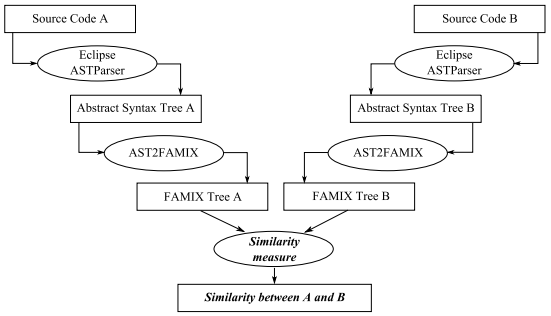


Figure 1: The figure shows the preprocessing steps of Java source code into FAMIX trees which are used as inputs for the similarity measures.

similarity between two or more Java classes: first, it transforms the abstract syntax tree representations of the classes' source code into intermediary FAMIX tree representations, and second, the similarity between these trees is computed based on tree similarity algorithms. The remainder of this section explains both steps in detail.

2.1 Tree Generation

Every piece of code can be represented as an abstract syntax tree (AST). An original, complete AST gets transformed into an abridged FAMIX tree representation using Eclipse's *ASTParser* and our *AST2FAMIX* converter as illustrated in Figure 1. We succinctly explain both of those tools.

The ASTParser is a component of the Eclipse JDT API that processes Java source code into its abstract syntax tree representation.[1] The classes that make up the tree are specified in the package `org.eclipse.jdt.core.dom`. By default, the ASTParser returns complete ASTs out of which the code can be perfectly reconstructed.

Our AST2FAMIX parser traverses the abstract syntax tree as generated by the ASTParser and builds a FAMIX representation from the nodes of the tree. Figure 2 shows a sample Java source code fragment and its corresponding FAMIX tree representation. The elements from the abstract syntax tree are mapped to FAMIX elements according to Table 1. Note that FAMIX does not represent all elements of Java abstract syntax trees, but instead represents a language-independent reduction of complete ASTs. This results in an information loss when converting Java to FAMIX since the level of granularity is reduced in the FAMIX model. However, the benefit from using FAMIX is the ability to perform programming language-independent similarity analyses, i.e., to compare, for instance, classes in C++ or Smalltalk with classes in Java. After the tree generation phase, the resulting FAMIX trees are passed to the similarity measures, which we discuss in the next subsection.

2.2 Tree Similarity Analysis

The current implementation of Coogle is able to detect structural similarity, i.e., similarity between the structure of the FAMIX trees of the source code. When representing source code as trees one important question arises: Is the order of the trees important? A Java compiler, for example, does not necessarily consider the order of top-level class body entities, such as methods or field declarations,

[1] `http://www.eclipse.org/jdt/`

```
package test;

import java.awt.Color;

public class Sample extends Parent {
  private String space = " ";

  public Sample(int input) {
    System.out.println(Color.BLACK);
  }
}
```

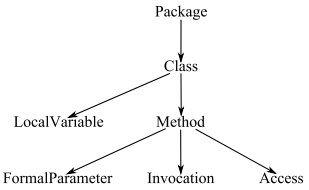


Figure 2: Original Java class source code and its corresponding FAMIX tree representation.

AST Node	*FAMIX Element*
-	FAMIXInstance
-	Model
PackageDeclaration	Package
TypeDeclaration	Class
-	InheritanceDefinition
FieldDeclaration	Attribute
MethodDeclaration	Method
SingleVariableDeclaration	FormalParameter
SingleVariableDeclaration	LocalVariable
ConstructorInvocation, SuperConstructorInvocation, ClassInstanceCreation, MethodInvocation, SuperMethodInvocation	Invocation
FieldAccess, SuperFieldAccess, SimpleName, QualifiedName	Access

Table 1: Eclipse AST elements with the corresponding FAMIX elements.

as relevant, whereas instructions in the bodies of these entities depend on the order of appearance in the source code. Hence, we need an algorithm that disregards the order between top-level entities but takes the order of low-level entities, such as method bodies, into consideration. As the FAMIX model does not represent all instructions which can occur in the body of an entity, e.g., it does not represent control structures, we would prefer using algorithms that perform a matching of unordered trees. Unfortunately, not all of the similarity measure algorithms that we have chosen have efficient solutions for unordered trees. For example, an unordered solution for the tree edit distance (see 2.2.1) is NP-complete as shown in [18]. We, therefore, implemented unordered tree matching for the bottom-up maximum common subtree search only and otherwise use algorithms for ordered trees.

2.2.1 Tree Similarity Algorithms

The literature on tree searching/editing is very elaborate. Shasha et al., for instance, describes his work on general tree and graph searching using exact and approximate search algorithms in [12]. Wang et al. presents a tool called *TreeRank* that does a nearest neighbor search for detecting similar patterns in a given phylogenetic tree [17]. These algorithms are highly specialized/optimized and, therefore, complex. Valiente [16], in contrast, discusses a number of standard tree searching and editing algorithms in detail providing efficient code implementation examples. To ensure a

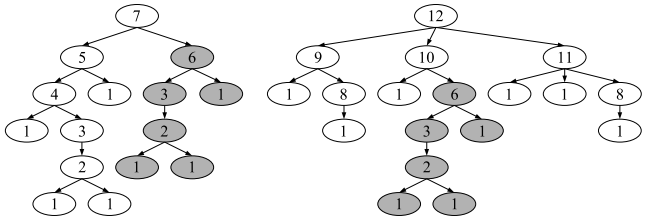


Figure 3: Bottom-up maximum common subtree isomorphism for two ordered trees (adapted from Fig. 4.15 in [16], page 225).

quick prototyping approach we decided to first implement three different algorithms from Valiente's work for measuring tree similarity: bottom-up maximum common subtree isomorphism, top-down maximum common subtree isomorphism, and tree edit distance.

Bottom-up Maximum Common Subtree Isomorphism. The goal of this algorithm is to find the largest isomorphic subtree of two given trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$. Valiente reduces this problem to the problem of partitioning the nodes $V_1 \cup V_2$ into equivalence classes. If two nodes v and w belong to the same equivalence class, the bottom-up subtree of T_1 rooted at node $v \in V_1$ is isomorphic to the bottom-up subtree of T_2 rooted at node $w \in V_2$. The equivalence classes of two ordered trees are illustrated by the numbers in the nodes in Figure 3, where the bottom-up maximum common subtree for the trees is highlighted in gray. We determine the isomorphism code of a given node by recursively building an isomorphism string consisting of the isomorphism codes of all children of the node. This string gets then compared to a collection of existing isomorphism strings. If the string is already in the collection, the corresponding equivalence class is read from the collection. If the isomorphism string is not contained in the collection, we add it to the collection and assign a new equivalence class code to the string. After collecting the equivalence classes of both trees T_1 and T_2, the algorithm searches for the biggest equivalence class by using a queue with the size of the nodes as priority. The first element in the queue is the node with the biggest size. This ensures that the matched subtree is indeed a maximum common subtree.

Valiente describes this algorithm for unlabeled trees only. We extended the algorithm to use labeled trees by assigning a unique integer value to each FAMIX node type (Package, Class, Method, etc.). The equivalence classes are then matched based on this value and the already defined equivalence class code. This solution is also suggested in [15]. We implemented the comparator pattern [3] for this label comparison.

To use this algorithm for unordered trees as well, the isomorphism codes of the children of a processed node are sorted based on their assigned FAMIX node type before searching for already existing code sequences in the equivalence class collection. This ensures that all children of a node only differing in order are treated the same, thus unordered.

Note that so far, we have only identified the maximum common subtree of both of the input trees. In order to get a similarity score between the two trees, we apply the following procedure: the size (number of nodes) of the first input tree T_1 is denoted by $|V_1|$. The cardinality $|V_2|$ stands for the size of the tree T_2 representing the second class. Furthermore, T_m denotes the maximum matched subtree of size $|V_m|$. An efficient similarity measure needs to satisfy the following properties: first, the more of T_1 is matched, the higher the similarity score of T_1 and T_2 is. This is expressed by $\frac{|V_m|}{|V_1|}$. This results in low values for complete matches of T_2, e.g., in the case if T_2 is much smaller than T_1. Second, complete matches should get higher values than non-complete ones, i.e., not the whole tree T_2 can be matched to T_1. We experimented with different possibilities. Finally, we decided to use a solution also described in [1] that results in a similarity value between 0 and 1 (1 for identical trees T_1 and T_2).

$$sim_{MaxCommonSubtree}(T_1, T_2) = \frac{2 \times |V_m|}{|V_1| + |V_2|} \qquad (1)$$

Having two trees T_1 and T_2 with $|V_1|$ and $|V_2|$ nodes, where $|V_1| \leq |V_2|$, the algorithm for ordered trees runs in $O(|V_1| \log |V_2|)$ time using $O(|V_1| + |V_2|)$ additional space (see Theorem 4.56 in [16]). The algorithm for unordered trees takes $O((|V_1|+|V_2|)^2)$ time and also uses $O(|V_1|+|V_2|)$ additional space (Theorem 4.60 in [16]).

Top-down Maximum Common Subtree Isomorphism. An algorithm to find a top-down maximum common subtree isomorphism for ordered and unordered trees is defined in [16]. This algorithm finds the largest common subtree of two given trees T_1 and T_2 under the prerequisite that the root of the common subtree is identical (same node type) with the root nodes of the compared trees. The differences between the algorithm for ordered trees and the algorithm for unordered trees are fundamental. For this paper, we implemented the algorithm for ordered tree matching.

Starting from the root nodes of T_1 and T_2, the algorithm recursively processes all children in preorder and compares each pair of nodes for equality. If two nodes match, they are added to a mapping $M \subseteq V_1 \times V_2$ that contains the complete subtree after the recursion finishes. Note that the recursion stops at nodes which do not match, i.e., the children of non-matching nodes are not getting compared to each other.

The comparison of the nodes during the recursive processing again allows for an extension of the algorithm to labeled trees, returning a successful match only when the labels (node types) match. Again, we used Equation 1 to get a similarity score from the size of the maximum common subtree and the two trees T_1 and T_2 under comparison.

This algorithm is very efficient with a running time of $O(|V_1|)$ and $O(|V_1|)$ additional space for two ordered trees T_1 and T_2, where $|V_1| \leq |V_2|$ (see Lemma 4.52 in [16]).

Tree Edit Distance. Calculating the tree edit distance is a completely different approach for tree analysis than the maximum common subtree isomorphism algorithms. The tree edit distance algorithm answers the question of how many steps it takes to transform one tree into another tree by applying a set of elementary edit operations to the trees: insertion, substitution, and deletion of nodes. For the ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ we denote a *deletion* of a leaf node $v \in V_1$ by $v \mapsto \lambda$ or (v, λ). The *substitution* of a node $w \in V_2$ by a node $v \in V_1$ is denoted by $v \mapsto w$ or (v, w) and an *insertion* of a node $w \in V_2$ as a new leaf into T_2 is de-

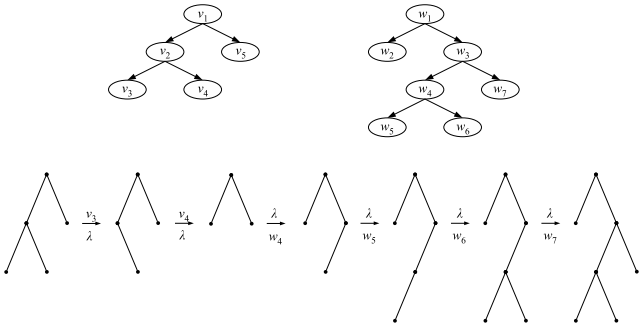


Figure 4: Transformation between two ordered trees (adapted from Fig. 2.1 in [16], page 56).

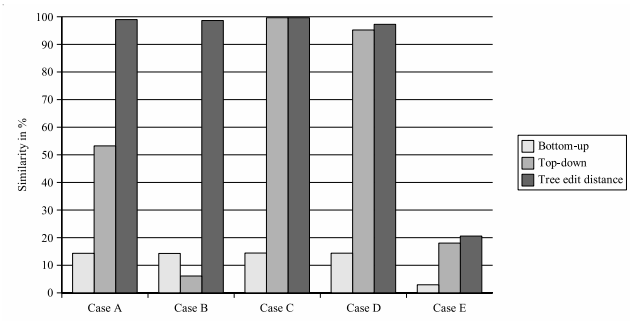


Figure 5: Results with constructed test cases for each similarity measure.

noted by $\lambda \mapsto w$ or (λ, w). Deletion and insertion operations are performed on leaves only. When deleting a non-leaf node v, every node in the subtree rooted at v has to be deleted first. The same applies to the insertion of non-leaves. A tree is transformed into another tree by using a sequence of elementary edit operations as illustrated in Figure 4. Note that in this figure, substitution of corresponding nodes is not indicated. The complete transformation script is: $[(v_1, w_1), (v_2, w_2), (v_3, \lambda), (v_4, \lambda), (v_5, w_3), (\lambda, w_4), (\lambda, w_5), (\lambda, w_6), (\lambda, w_7)]$. Costs are assigned to all elementary edit operations. Our current implementation uses a cost function of $\gamma(v, w) = 1$ if $v = \lambda$ or $w = \lambda$ and $\gamma(v, w) = 0$ otherwise. The function reflects that node substitutions usually denote relabelings which have little structural significance and should, therefore, not be weighted. The edit distance then is the least-cost transformation of T_1 to T_2 normalized by the sum of nodes in T_1 and T_2. The lower the normalized edit distance of two trees, the higher their similarity.

$$sim_{TreeDistance}(T_1, T_2) = \frac{TreeDist(T_1, T_2)}{|V_1| + |V_2|} \qquad (2)$$

Finding the least-cost transformation of an ordered tree T_1 and T_2 by determining shortest paths in an edit graph runs in $O(|V_1||V_2|)$ time using $O(|V_1||V_2|)$ additional space (see Lemma 2.20 in [16]).

3. EXPERIMENTAL EVALUATION

To evaluate the ability of our approach to detect similar entities in a software project, we ran the evaluation on two datasets. First, we constructed a set of special test cases capturing frequently occurring changes which happen during software development. Evaluating our similarity detection algorithms for these constructed changes, we were able to analyze how specific changes affect structural similarity. In a second experiment we chose a well known Java project as the dataset for our implemented similarity measures. We used Eclipse's compare plug-in `org.eclipse.compare`[2] and measured the similarity of the classes within the same version as well as between different versions of the plug-in. This analysis focused on the efficiency of the measures for detecting structural similarities between two classes in the former and on software evolution considerations in the latter case. The remainder of this section shows our obtained results of this two experiments.

[2]`http://dev.eclipse.org`

3.1 Experiment #1: Constructed Test Cases

As a basis for the construction of the test cases we took the class `AzureusCoreImpl` from the Azureus project (except in test case E).[3] This class was chosen as it uses both "normal" and `static` attributes and methods. Additionally, it defines getter and setter methods for its attributes. In addition to the test cases we compared `AzureusCoreImpl` with an empty class to ensure the plausibility of our implemented measures. The defined test cases are the following:

Test Case A: Add Constructor. This test case adds a new constructor with a single `this()`-invocation as body to the class.

Test Case B: Add Attribute. We add a new attribute with its respective getter and setter methods to the class. The rest of the class is left unchanged.

Test Case C: Add Invocation. We insert an invocation, i.e., a method call into the body of an existing method.

Test Case D: Method Extraction. This case models the movement of code statements from an existing method into a new method. An invocation of the new method is added to the original one. This change often happens during a code refactoring to remove duplicated code or when pulling-up code into parents [2].

Test Case E: Implement Interface. The interface programming pattern is one of the most important design patterns in object-oriented programming [3]. This test case measures the similarity between classes implementing the same interface. We expect this case to have very low structural similarity, as the structural similarities between classes implementing the same interfaces are limited to the implementation of the methods defined in the interface, which may be implemented in structurally completely dissimilar ways.

3.1.1 Results of Test Cases

This section presents our findings, discusses major drawbacks, and details the performance of the implemented algorithms. The results of this experiment are shown in Figure 5. Confirming our expectations, none of the algorithms

[3]`http://azureus.sourceforge.net`

detected any significant similarity in Test Case E. A similarity measure finding classes implementing the same interface would have to put special emphasis on the interface definitions, which none of our measures does.

Bottom-up Maximum Common Subtree Isomorphism. The obtained results show clearly that the bottom-up subtree isomorphism algorithm performs worst for detecting similar Java classes. The similarity score remains at about 14%. The reason for this is that this measure uses equivalence classes for checking equality of different nodes. For a better match, the measure would have to include the unchanged parts of the tree as well, requiring the equivalence class of the root to remain the same. This is not the case in our tests as a node insertion/deletion at tree depth level 1 changes the equivalence code of the root. Hence, the measure matches the biggest subtree from level 2, which usually is the biggest method.

Top-down Maximum Common Subtree Isomorphism. We obtain mixed results with this measure. Case C, D, and partly case A show good scores for detecting similarity with the top-down algorithm. In case B, similarity is not well detected because the insertion of a variable stops the matching process too early. Hence, this algorithm overvalues small changes near the root, such as simple insertions, deletions, and relabelings. This limitation could possibly be lessened by continuing the comparison even when nodes have different names or by using a top-down algorithm for unordered trees.

Tree Edit Distance. The tree edit distance algorithm performed best for our test cases. The similarity scores in each test (except E) are over 97%, which is sufficient for establishing a similarity relation between two classes with a high accuracy. The big advantage of this algorithm in comparison to the two maximum common subtree algorithms is that it is not as susceptible to node insertions/deletions.

3.2 Experiment #2: Java Project

The `org.eclipse.compare`-plug-in of Eclipse is used as sample, real-world Java project to test the similarity measures. The analysis demonstrates the ability of using the implemented similarity measures in a non-laboratory environment and critically highlights shortcomings. "The plug-in provides support for performing structural and textual compare operations on arbitrary data" (from its Javadoc). The goal of this experiment was to visualize the project's code evolution steps by comparing different versions of the plug-in. I.e., how strongly is class similarity affected when changes/refactorings occur to the code from one version to the next. To achieve this goal, we have compiled a heatmap illustrated in Figure 6 showing similarities between classes of two different versions of the plug-in. For our experiment, we used versions 3.0 and 3.1 of the compare-plug-in.

3.2.1 Results of org.eclipse.compare-Plug-in

Based on our findings of Experiment #1 (see Section 3.1), we chose to perform a similarity analysis using the tree edit distance measure to determine class similarity within the `org.eclipse.compare`-plug-in. Consider the heatmap depicted in Figure 6: class similarity is computed between any class of version 3.0 (on the x-axis) and 3.1 (on the y-axis).

Figure 6: Heatmap showing similarities between all classes of versions 3.0 and 3.1 of the `org.eclipse.compare`-plug-in. Black squares indicate similarity above 90%, dark-gray above 75%, and light-gray above 50%.

The similarity score between two classes is visualized as a shaded square. Similarity above 90% is indicated by black squares. Squares shaded in dark-gray denote class similarity above 75%, whereas squares in light-gray indicate similarity above 50%. Similarities below 50% are not shaded.

The elements on the diagonal indicate how strong the software has changed between versions. I.e., the diagonal is clearly visible as a dark "line" showing that the same classes of two versions have very high similarity—more than 75% in most cases. As shown in Figure 6, a few classes have changed from version 3.0 to 3.1. As an example, we picked the class `MergeMessages` from the package `org.eclipse.compare.internal.merge` (indicated in Figure 6 by a circle on the line marked with the arrow labeled with a "2") that has a similarity of 74% between the two versions. Examining the source code of the two versions, we found that three more static fields were added and one method was removed and replaced with a static initializer in version 3.1. Hence, three more attribute nodes and one method node with an invocation node are added to the tree representation in version 3.1 resulting in a similarity score of 74%.

The map of Figure 6 is not only useful to visualize software evolution but also to measure and visualize similarity between any two classes of a software project. Again, we explain this with an example: "The interface `ICompareNavigator` is used to navigate through the individual differences of a `CompareEditorInput`" (from its Javadoc). It is a simple interface defining exactly one method with one argument. Similarity between this interface and 25 other classes/interfaces out of 114 is very high, i.e., above 75% (refer to Figure 6 that shows an arrow labeled with a "1" pointing to the line corresponding to `ICompareNavigator` in this

heatmap). The entities with high similarity are, for instance, `INavigatable` (sim=75%) also specifying one method with one parameter but, in addition, defining a static field. Hence, the corresponding tree representations of these interfaces are very similar, which is correctly reflected by their high similarity value. Another interface similar to `ICompareNavigator` is `IViewerDescription` (sim=75%) also defining one method but with 3 arguments instead of 1 as in the case of `ICompareNavigator`. Note that Figure 6 shows the similarities of classes in version 3.0 with those in 3.1. To compare the similarity of classes within the same version, we double-checked our findings in a heatmap comparing classes from version 3.0 with all other classes from version 3.0 (omitted due to space constraints). Naturally, all similarities on the diagonal in this within-version analysis were 100% equal since classes where compared with themselves. The line comparing `ICompareNavigator` with the other classes within the package showed a similar (but not the same) behavior as in the between-version analysis above.

4. DISCUSSION

In this section we discuss the results of our experiments, highlight shortcomings of the tree similarity algorithms and propose possible improvements for future work.

4.1 Comparison of Implemented Measures

Our obtained results showed that a bottom-up maximum common subtree isomorphism match is not a good measure for similarity when evaluated on the user-constructed test cases. It is too susceptible to subtle code modifications in methods, which usually cause changes at the bottom-level of the tree.

The top-down maximum common subtree algorithm produces promising results on the test cases. The measure is a good indicator for similarity as it is able to detect classes with similar structures. One negative characteristic of this algorithm is that simple changes at the top of the tree, such as adding a new attribute or inserting an attribute between existing methods, reduces the reliability of the measure. The top-down approach is, however, not as sensitive to changes at the bottom of the trees as the bottom-up approach.

The best-performing similarity algorithm turned out to be the tree edit distance on both datasets. It detected the small changes specified in the test cases and provided itself as a good indicator for structural similarity when tested on the compare-plug-in. Note that the price to pay for it is high: it runs in quadratic time complexity $O(|V|^2)$ of its tree input size, which resulted for the $n \times n$-comparison of 114 classes of the `org.eclipse.compare`-plug-in in more than an hour to finish.

4.2 Limitations and Future Work

Field or Method Name Matching. Additional information can be gained from class, field, or method names in similarity comparisons. Methods and fields used for similar tasks are (hopefully) named with similar names if the code was created abiding reasonable naming standards. This helps detecting cloned parts of classes. Text-based similarity measures can then be used to calculate similarity between names. The current implementation of Coogle treats nodes representing `class X` and `class Y`, for instance, as equal since only the types ("Class" in this case) of nodes are compared. Note that because Coogle currently does not compare names, also access control qualifiers, such as `public`, `protected`, and `private`, are not taken into consideration since those are not proper FAMIX entities, but simple string attributes of entities.

FAMIX Limitations. The FAMIX model represents a fixed set of elements (see Table 1), i.e., invocations, declarations, attributes, etc., but does not include assignments, mathematical operations, and control structures. This limits the detection of small changes to basic, top-level instructions. Different results are to be expected when bypassing FAMIX and directly generating the comparisons from complete abstract syntax trees or when extending the FAMIX model with a more fine-grained hierarchical structure. This might help to detect "real" functionally similar classes and diminish the detection of classes only structurally similar. On the other hand, because the size of the input trees would be much bigger, the algorithm's performance would get worse.

Surrounding String Matching. We plan to include surrounding text in similarity comparisons, as statements in source code are frequently surrounded by free text, such as Javadoc. Analyzing the similarity of this text and including this textual similarity in the measures will probably further boost the precision of the similarity algorithms.

Similarity Measure Combinations. We think it is useful to investigate different combination approaches of structure-based as well as text-based similarity measures to further increase the precision of matches. We, therefore, postponed the implementation and evaluation of such combinations to the future.

5. RELATED WORK

Both Mishne & Rijke [7] and Neamtiu et al. [9] define a conceptual model for source code representation that partially resembles the abstract syntax tree as defined by Eclipse. Mishne & Rijke use code similarity for retrieving similar code fragments from an existing repository of code documents based on classifying instructions in the code with varying weights. They do, however, not apply tree similarity measures to retrieve similar code fragments from the repository. Neamtiu et al. extract similarity by mapping corresponding AST elements of two code documents. Again, this algorithm does not use a generic tree similarity algorithm. Their focus lies on the mapping algorithms to measure similarity between AST representations, which relies on node names within the ASTs.

A different approach is introduced by Kontogiannis who defines a Program Description Tree (PDT) that is generated from code fragments [5]. These fragments are treated as behavioral entities, i.e, as independent components interacting with resources and other entities of the software project. The PDT, therefore, not only represents structural information like an AST does, but also contains information such as interactions and read or write accesses, i.e., behavioral information. Similar fragments are detected by searching for entities with similar characteristics of these PDTs. In our current implementation of Coogle we do not examine attribute accesses, but take interactions, such as method invocations, into account. It would be interesting to extend

Coogle to operate on PDTs and compare the performance of the two.

A similar approach to ours is described by Holmes & Murphy in [4]. Their tool, *Strathcona*, is used to find source code in an example repository by matching the code a developer is currently writing. In contrast to Coogle, this approach is not based on tree similarity algorithms, but on multiple structural matching heuristics, such as examining inheritance relationships, method calls, and class instantiations. The measures are applied to the currently typed code. Matched examples from the repository are retrieved and displayed to the developer for selection. We have not yet implemented such a feature in Coogle, which could clearly help to foster code reuse. Again, it would be interesting to compare their approach that makes heavy use of domain knowledge (in the form of heuristics) with ours that solely relies on the power of the similarity algorithms.

A promising approach to help developers navigate source code efficiently is presented by Robillard in [11]. The presented technique is able to find relevant (in our context similar) elements in source code to a given query element by examining the topological properties of the structural dependencies of the query element. Elements are considered as relevant if they fulfill well-defined criteria, such as specificity and reinforcement to other elements. In that context, it would be interesting to know about the performance of our implemented tree similarity algorithms as another criteria to determine relevant elements of interest.

Finally, various other approaches exist for detecting similarity in trees and source code. Baxter & Manber describe a tool that analyses projects for duplicated code [1]. Their implemented algorithm is based on abstract syntax trees and employs a hashing function on code fragments for detecting exact and near-miss clones. Myles & Collberg take a similar approach by using a birth-marking technique, deducing unique characteristics from the instruction set of a program to detect software theft [8]. While it would make perfectly sense to use Coogle as a tool to find code clones and possible software plagiarisms, we currently only employ it to find similarity between classes to, for instance, comprehend software evolution steps.

6. CONCLUSION

In this paper we presented our approach to detect similarities between different Java classes based on abstract syntax trees. Similarity is calculated by means of three tree similarity algorithms: bottom-up maximum common subtree isomorphism, top-down maximum common subtree isomorphism, and the tree edit distance. We chose the FAMIX model of object-oriented programming languages to represent the compared trees.

The trees under comparison are generated in two steps: first, the abstract syntax tree representation of the source code is built through Eclipse's ASTParser, and second, our AST2FAMIX parser traverses the abstract syntax tree and builds a FAMIX representation from the nodes of the tree. Finally, the similarity algorithms are used to determine the degree of similarity between different FAMIX trees.

We validated our approach on two datasets: user-constructed test cases and the `org.eclipse.compare`-plug-in. Of the three tree similarity measures, we found the tree edit distance to produce the best results, followed by the top-down maximum common subtree isomorphism algorithm. Especially, when measuring the effects of small changes, the tree edit distance measure proved to be very robust. When evaluating our approach on the `org.eclipse.compare`-plug-in, we successfully detected structural similarities between classes of the same version. In addition, our approach proved to be very promising when comparing different versions of the project, i.e., when focusing on questions concerning software evolution, for instance, how did the classes change from one release to the next. Visualizing class similarities by the use of heatmaps, we were able to illustrate this code evolution within the `org.eclipse.compare`-plug-in.

As our initial results have shown, our approach is very promising. It (1) indeed identified similar Java classes, (2) successfully identified the ex ante and ex post versions of refactored classes, and (3) provided some interesting insights into the within-version and between-version dependencies of classes within a medium-sized Java project.

7. REFERENCES

[1] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke,and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[4] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125, 2005.

[5] K. Kontogiannis. Program Representation and Behavioural Matching for Localizing Similar Code Fragments. In *Proceedings of the 1993 Conf. of the Center for Advanced Studies on Collaborative Research*, pages 194–205, 1993.

[6] A. Michail and D. Notkin. Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. In *Proceedings of the 21st International Conference on Software Engineering*, pages 463–472, 1999.

[7] G. Mishne and M. de Rijke. *Source Code Retrieval using Conceptual Similarity*. 2004.

[8] G. Myles and C. Collberg. K-Gram Based Software Birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 314–318, 2005.

[9] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[10] Object Technology International, Inc. Eclipse Platform Technical Overview. 2003.

[11] M. P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *Proceedings of the 10th European Software Engineering Conference*, pages 11–20, 2005.

[12] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 39–52, 2002.

[13] S. Tichelaar. *FAMIX Java Language Plug-in 1.0*. 1999.

[14] S. Tichelaar, P. Steyaert, and S. Demeyer. *FAMIX 2.0: The FAMOOS Information Exchange Model*. 1999.

[15] G. Valiente. Simple and Efficient Tree Pattern Matching. Technical Report LSI-00-72-R, Technical University of Catalonia, Dec. 2000.

[16] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002.

[17] J. T.-L. Wang, H. Shan, D. Shasha, and W. H. Piel. TreeRank: A Similarity Measure for Nearest Neighbor Searching in Phylogenetic Databases. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, pages 171–180, 2003.

[18] K. Zhang, R. Statman, and D. Shasha. On The Editing Distance Between Unordered Labeled Trees. *Information Processing Letters*, 42(3):133–139, 1992.

Mining Version Archives for Co-changed Lines

Thomas Zimmermann[1] Sunghun Kim[2] Andreas Zeller[1] E. James Whitehead Jr.[2]

[1] Department of Computer Science
Saarland University
Saarbrücken, Germany
{tz, zeller}@acm.org

[2] Department of Computer Science
University of California
Santa Cruz, CA, USA
{hunkim, ejw}@cs.ucsc.edu

ABSTRACT

Files, classes, or methods have frequently been investigated in recent research on co-change. In this paper, we present a first study at the level of lines. To identify line changes across several versions, we define the annotation graph which captures how lines evolve over time. The annotation graph provides more fine-grained software evolution information such as life cycles of each line and related changes: "Whenever a developer changed line 1 of version.txt she also changed line 25 of Library.java."

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*corrections*, *version control*; D.2.9 [**Management**]: Software configuration management

General Terms

Management, Measurement

1. INTRODUCTION

One of the most frequently used techniques for mining version archives is *co-change*. The basic idea is that items that are changed together, are related to each other. These items can be of any granularity; in the past co-change has been applied to changes in modules [7], files [2], classes [8], and methods [14]. All these approaches stopped at the granularity of methods. Applying them to more fine-grained items such as blocks or lines seemed infeasible, in particular since they are difficult to identify across versions.

Typically lines are identified by their line number. However, since lines may be moved within files, e.g., when other lines are inserted or deleted before, line numbers are not fixed across versions and thus not suitable as identifiers for co-change analysis. We abstract line evolution from line numbers by representing each line as several nodes in a graph (one node for each revision); edges connect lines (nodes) that evolved from each other. We call this graph an *annotation graph* (Section 2).

Today, many SCM systems such as CVS and Subversion come with an annotation feature that returns for each line the last modification. Such information is not enough to track lines across revisions. In contrast, using the annotation graph we can build more general annotation algorithms that return *all* past modifications instead of just the last one (Section 3). Such annotations provide information about the life cycle of lines (Section 4).

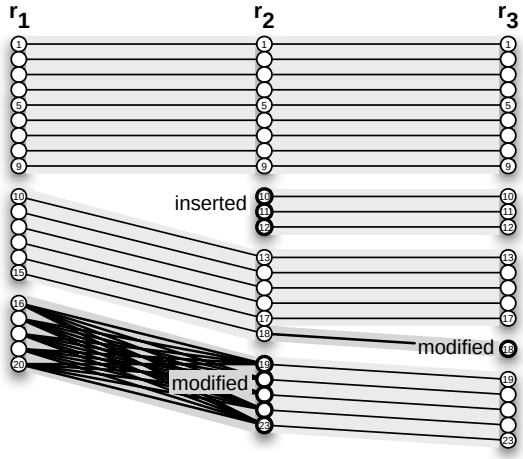


Figure 1: Tracking lines with the annotation graph.

In recent research, data mining on co-change information was used to recommend related locations such as files [13] and methods [16] after one initial change. In Section 5 we show that this is also possible for lines: "Whenever a developer changed line 1 of version.txt she also changed line 25 of Library.java." In Section 6 we discuss related work and Section 7 closes the paper with an outlook on future work.

2. TRACKING LINES

Tracking how lines evolve over time requires the identification of lines *across several versions* of a file. Within one single version, lines are typically identified by line numbers or in some cases by their contents. However both cases do not work when applied to several versions: line numbers may change when other lines are deleted or inserted, and the content of lines may be modified.

2.1 What are Annotation Graphs?

To capture how lines evolve over time, we introduce the annotation graph. The annotation graph is a multipartite graph where each part corresponds to one version of a file. Within each part/version every line is represented by a single node; edges between node indicate that a line originates from another: either by modification or by movement. Whether a line was changed in a revision is captured by labels, e.g., bold nodes indicate changes lines.

MSR'06, May 22–23, 2006, Shanghai, China.

As an example consider Figure 1 which represents several changes in an annotation graph. Edges connect lines that relate to each other across revisions, e.g., line 1 in revisions r_1, r_2, and r_3. Modifications such as from lines 16–20 in r_1 to lines 19–23 in r_2 result in a complete bipartite subgraph for that area. In other words, every node from 16 to 20 in r_1 is connected with every node from 19 to 23 in r_2.

Formally, an annotation graph $G = (V, E)$ for a file with n revisions $r_1, \ldots, r_n$ (sorted by their creation time) consists of nodes

$$V = \bigcup_{i=1}^{n} \{(r_i, m) \mid m \in \{1, \ldots, \mathit{number\_of\_lines}(r_i)\}\}$$

and edges $e = ((r_a, l_a), (r_b, l_b)) \in E$ for which

1. r_b is a direct successor of r_a and
2. l_b originates from l_a—either by modification (contents differ) or by movement (contents and relative position are equal)

Additionally, when lines were changed, we label the corresponding nodes with a description of the change such as the author who changed the lines, or the transaction in which the lines were changed.

2.2 How to Read GNU's diff

In order to construct an annotation graph, we need to compare all subsequent revisions of a file. For computing textual differences, we use the GNU *diff* tool. The *diff* tool returns a list of regions that differ in the two files; each region is called a *hunk*. Basically, there are three different kinds of changes:

Modifications. In an annotation graph, modifications result in a complete bipartite subgraph like in Figure 1 between lines 16–20 in revision r_1 and lines 19–23 in revision r_2.

Additions. For the annotation graph, additions do not result in any edges, only the positions of following lines are updated. In Figure 1, the lines 10–12 are inserted in revision r_2, thus line 10 of revision r_1 corresponds to line 13 in r_2.

Deletions. For the annotation graph, deletions do not result in any edges, only the positions of following lines are updated.

When comparing two text files with *diff*, we specified the *–text*, *–minimal*, and *–strip-trailing-cr* options. These options turned out to be very effective to return a small set of differences and to address the *carriage return* problem that *diff* and CVS suffer from.

2.3 How to Compute Annotation Graphs

Once we have computed the changeregions for all subsequent revisions, we can use this information to build an annotation graph for a file. When computing an annotation graph, one can either start from the first revision computing forward (to the last revision), or start from the last revision computing backward. We now describe a *forward-directed* algorithm that starts with the first revision; fore more details we refer to the extended version of this paper [15].

First the algorithm creates nodes for each revision and each line with the method *createNode*. Next, it iterates over all pairs (*revL*, *revR*) of subsequent revisions. For each pair it computes the differences (hunks) between *revL* and *revR* which then are sorted by their position *R_from* in the later revision *revR*. These hunks are then processed to create edges between nodes:

– for unchanged lines exactly one edge between the matching lines *posL* and *posR*;

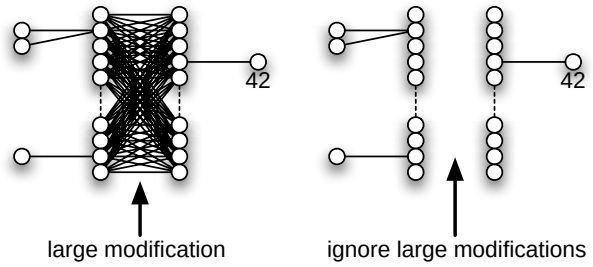


Figure 2: Ignoring large modifications for annotation graphs.

– for modified lines all possible edges, which means *posL* $\in \{L\_from \ldots, L\_to\}$ and *posR* $\in \{R\_from \ldots, R\_to\}$;

– for inserted and deleted lines no edges are created.

For modifications and additions, we label the nodes of the later revision *revR* with information about the change, such as author and transaction. These labels are later used to compute annotations that are more general than the ones provided by existing SCM systems (see Section 3).

2.4 How to Recognize Large Modifications

One problem for annotation graphs are changes that *modify large* parts of a file, since they results in a large number of edges. As an example consider the left part of Figure 2. When we investigate the evolution of line 42 and go back in time, we come across a large modification. If we take this modification into account, line 42 originates from every modified line. Such a result is not reasonable for evolution analysis.

In order to reduce noise, we treat large modifications not as a modifications but as combined deletions and additions. This means that for large modifications, we do not create any edges in the annotation graph (see the sketch in the right part of Figure 2).

For recognizing large modifications we use a heuristic. Let length_L and length_R be the lengths of the left (L) and right (R) region of a hunk `fct`, and $\mathit{file\_length}_L$ and $\mathit{file\_length}_R$ be the lengths of the corresponding files. A hunk is a large modification if one of the following conditions hold:

– *Region lengths exceed a threshold*

$$\begin{aligned} &\mathit{length}_L > \mathit{max}(\alpha \cdot \mathit{file\_length}_L; \beta) \\ \vee\ &\mathit{length}_R > \mathit{max}(\alpha \cdot \mathit{file\_length}_R; \beta) \end{aligned}$$

– *Ratio of region lengths exceeds a threshold*

$$\frac{\mathit{length}_L}{\mathit{length}_R} < \frac{1}{\gamma} \ \vee\ \gamma < \frac{\mathit{length}_L}{\mathit{length}_R}$$

The first condition recognizes changes that affect large parts of a file, in contrast, the second one recognizes changes that insert or delete large portions to or from a region. For our experiments, we used $\alpha = 0.10$ and $\beta = \gamma = 4$.

3. ANNOTATING LINES

Most SCM systems come with an annotation feature that returns for each line when it was inserted and by whom. For instance, the CVS annotations in Figure 3 for revision 1.17 of file Foo.java, tell us that line 39 was inserted by Mary in revision 1.10 and line 40 was inserted by Kate in revision 1.14. In this section, we briefly show how to compute such annotations using the annotation graph. While SCM systems typically return only information about the *last* change, the annotation graph can provide more general annotations that collect information about *all* past changes.

```
   $ cvs annotate -r 1.17 Foo.java
   ...
19: 1.11 (john 12-Feb-03): public int a() {
20: 1.11 (john 12-Feb-03):     return i/0;
   ...
39: 1.10 (mary 12-Jan-03): public int b() {
40: 1.14 (kate 23-May-03):     return 42;
   ...
59: 1.10 (mary 17-Jan-03): public void c() {
60: 1.16 (mary 10-Jun-03):     int i=0;
   ...
```

Figure 3: CVS annotations for Foo.java

Annotating with the last change. When computing annotations for a revision r_s, we perform for each line l_s a backward-directed breadth-first search in the annotation graph, starting from node (r_s, l_s). The search stops when we visit a node (r_x, l_x) that is labeled as a change (either the line was added or modified). We then annotate the line l_s with information from revision r_x, such as the revision identifier, the author, or the time of the change. Note that for a line l_s the last change is unique, thus l_x and r_x are unique too. It may also hold that $r_s = r_x$ in case (r_s, l_s) is already labeled as a change.

Annotating with all changes. When annotating a revision r_s with all changes, we also perform for each line l_s a backward-directed breadth-first search in the annotation graph, starting from node (r_s, l_s). However, we do not stop when visiting a changed node; instead we collect for every visited node that is labeled as a change, its information in (multi)sets. Once the breadth-first search is completed, we annotate the line l_s with these sets.

4. LIFE CYCLE OF LINES

In order to investigate the life cycle of lines for the complete ECLIPSE project (snapshot 2005-11-23) we annotated all text files with information about *all* past changes. In particular, we collected the revision identifiers and the authors. Additionally, we ignored lines containing whitespace or single curly braces. Computing the annotations took approximately 10 hours for 31,950 files.[1] Using these annotations we provide answers to the following questions.

How frequently are lines changed? We computed for each line the *change count*, that is the number of distinct revisions in its annotation. Note that we also counted the addition of a line as a change. Figure 4 shows the distribution of the change count broken down to different file extensions. We observe that most lines are changed only one time, in other words, they are inserted to a file and never touched again. This is the case for almost every line in `.dtd` and `.txt` files. In contrast, lines in `.properties` files are more frequently modified (44% at least once). Such files are used to separate properties (e.g., text messages) from the actual ECLIPSE code.

How many developers change a line? see [15].

What are the most frequently changed lines? see [15].

5. FINDING RELATED LINES

In this section, we show how to compute related lines using frequent pattern mining. In order to create the input for data mining, we annotated all lines of ECLIPSE (snapshot 2005-11-23) with all past changes. However, instead of revision ids that are only unique per file, we used the corresponding transaction ids. As a result, we

[1] All experiments were run on an Opteron cluster using eight processors, each with 2 Mhz and 2 GB memory.

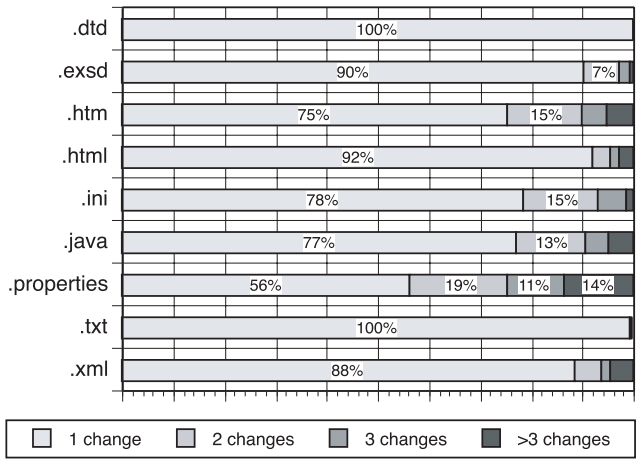


Figure 4: How frequently are lines changed?

get for every line the set of transactions that changed this line is the past. By using transactions instead of revisions, we are able to recognize patterns that are spread across several files.

For our experiments with frequent pattern mining, we used the Apriori algorithm [1]. In order to keep the complexity low, we applied the following optimizations:

- ignore lines containing whitespace or just a single curly brace
- investigate only modifications (not additions)
- combine lines with exactly the same change history to blocks and use blocks instead of lines as input for mining

Using the above optimizations, we could reduce the size of the input for data mining from 4,493,244 changes on lines to 255,778 changes on blocks and the calculation time to 19 seconds. On the new input we mined for all patterns that had a minimum support count of 23. The support count tells us how frequently lines that are part of a pattern have been changed together in the past. For lower support thresholds the computation did either not finish or ran out of memory (more than 16G). Improving the mining performance will remain future work.

Because of the high support count threshold we found only 29 patterns and only two them were interesting. The first pattern was found in file plugin.xml where several lines defining icons. These lines were changed together 23 times.

```
line 666:   icon="$nl$/icons/full/obj16/package_obj.gif"
     676:   icon="$nl$/icons/full/elcl16/static_co.gif"
     686:   icon="$nl$/icons/full/elcl16/constant_co.gif"
     717:   icon="$nl$/icons/full/obj16/package_obj.gif"
     727:   icon="$nl$/icons/full/elcl16/static_co.gif"
     737:   icon="$nl$/icons/full/elcl16/constant_co.gif"
     750:   hoverIcon="$nl$/icons/full/elcl16/exc_catch.gif"
     752:   disabledIcon="$nl$/icons/full/dlcl16/exc_catch.gif"
     753:   icon="$nl$/icons/full/elcl16/exc_catch.gif"
     762:   icon="$nl$/icons/full/obj16/package_obj.gif"
     776:   icon="$nl$/icons/full/obj16/package_obj.gif"
     808:   hoverIcon="$nl$/icons/full/etool16/run_sbook.gif"
     810:   disabledIcon="$nl$/icons/full/dtool16/run_sbook.gif"
     812:   icon="$nl$/icons/full/etool16/run_sbook.gif"
```

The second pattern was spread across three files: a text file version.txt, and two Java files, both named Library.java, but within different directories. The lines contain the minor version of an SWT component and were changed 171 times together.

```
version.txt              line 1:   version 3.215
j2me/.../Library.java, line 25: static int MINOR_VERSION = 215;
j2se/.../Library.java, line 25: static int MINOR_VERSION = 215;
```

Using the above pattern, we can infer association rules such as: "*Whenever a developer changed line 1 of version.txt she also changed line 25 of Library.java.*" Such a rule holds with a high confidence of 87% (171 out of 196 changes).

6. RELATED WORK

In this section we discuss work that is related to annotation graphs.

Annotating revisions. Chen et al. developed the CVSSearch tool that annotates source code with the log messages from the last code change and uses this information to guide programmers using *textual similarity* [5]. Hassan and Holt annotated static dependency graphs with *sticky notes*. A sticky note for a dependency contains the developer who created it, including the time when it was created and the log message that was provided with that change. In contrast to the work by Chen et al. and Hassan and Holt, the annotation graph considers *all* changes and not only the last ones.

Related changes. Ying et al. [13] and Zimmermann et al. [16] applied data mining on co-change information in order to recommend related locations such as files or methods. We applied the same data mining techniques, however, our focus was on lines and not on coarse-grained items such as methods or files.

Origin analysis. Godfrey et al. [9] and Kim et al. [10] proposed algorithms called origin analysis, which identify the same entities over revisions by computing entity similarities—even when entity name changes. Origin analysis is similar to our work in that origin analysis tries to map entities over revisions, while the annotation graph maps lines over revisions.

Small changes. Sliwerski et al. showed how to locate fix-inducing changes in version archives [12]. A subset of fix-inducing changes has been investigated under the name *dependencies* by Purushothaman and Perry [11] to measure the likelihood that small changes introduce errors. Their dependency concept is similar to the annotation graph, however our work focuses on the annotation of line evolution in order to compute related changes.

7. CONCLUSION

In this paper we presented the annotation graph which captures the evolution of lines. With this graph we carried out a first investigation of the life cycle of lines and pointed out that it is possible to find related lines with co-change analysis. However, data mining on co-changed lines is still expensive. Thus our future work will focus on improving the mining performance and exploring other mining techniques.

Origin analysis on lines. Modifications result in a complete bipartite subgraph, since we cannot figure out which lines are changed to which lines (see Section 2.2). We will apply origin analysis [9, 10] in the line level to identify the origin of each line. This will lead to more precise annotation graphs.

Large modifications. The parameters for recognizing large modifications (see Section 2.4) were selected after a manual inspection of several code changes. We are planning a sensitivity analysis to determine how our results depend on the selection of these parameters.

Increase mining performance. Frequent pattern mining on line level turned out to be too extensive. As a first optimization we combined lines that shared the same history to blocks. This yielded first results, however only for patterns with high support count values. Currently, we investigate other optimizations to find interesting patterns that have a low support.

Visualize evolution of lines. Using the models and layout algorithms implemented in EpoSee [4] and CCVisu [3] and fractal figures [6], we plan to visualize line level co-changes to identify related lines and to detect abnormalities.

Build tool support. We are currently developing plug-ins that will integrate annotation graphs into the ECLIPSE development environment. The user will be able to explore the evolution of lines with an *annotation graph browser* and related lines will be automatically displayed with tool tips.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of 20th International Conference on Very Large Data Bases (VLDB 1994)*, pages 487–499. Morgan Kaufmann, September 1994.

[2] J. Bevan and E. J. Whitehead Jr. Identification of software instabilities. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 134–145, Victoria, Canada, 2003. IEEE Computer Society.

[3] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, pages 259–268. IEEE Computer Society Press, Los Alamitos (CA), 2005.

[4] M. Burch, S. Diehl, and P. Weißgerber. Visual data mining in software archives. In *Proceedings of the 2005 ACM symposium on Software visualization (SoftVis 2005)*, pages 37–46, New York, NY, USA, 2005. ACM Press.

[5] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 364–373, Florence, Italy, 2001. IEEE Computer Society.

[6] M. D'Ambros, M. Lanza, and H. Gall. Fractal figures: Visualizing development effort for cvs entities. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 46–51. IEEE Computer Society, Sept. 2005.

[7] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 190–197, Bethesda, Maryland, USA, 1998. IEEE Computer Society.

[8] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Helsinki, Finland, 2003. IEEE Computer Society.

[9] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[10] S. Kim, K. Pan, and E. J. Whitehead Jr. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, pages 143–152, Pittsburgh, Pennsylvania, USA, 2005. IEEE Computer Society.

[11] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR 2005)*, St. Louis, Missouri, USA, 2005. ACM Press.

[13] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[14] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–84, Helsinki, Finland, 2003. IEEE Computer Society.

[15] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr. Mining version archives for co-changed lines. Technical report, Saarland University, Saarbrücken, Germany, March 2006. Available at http://www.st.cs.uni-sb.de/softevo/.

[16] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

Constructing Universal Version History

Hung-Fu Chang
University of Southern California
University Park Campus,
University of Southern California
Los Angeles, CA 90089
+1 213 740-9621
hungfuch@usc.edu

Audris Mockus
Avaya Labs Research
233 Mt. Airy Rd.
Basking Ridge, NJ, USA 07920
+1 908 696-5608
audris@avaya.com

ABSTRACT
Developers often copy code for parts or entire products to start a new product or a new release. In order to understand the software change history and to determine the code authorship, we propose to construct a universal version history from multiple version control repositories. To that end we create two practical code copy detection methods at the level of the source code file: prefix-postfix algorithm and prefix algorithm. The full pathname of a file and its version history are used to construct the universal version history of a file by linking together change histories of files that had the same code at any point in the past. The assumption of both algorithms is that developers often duplicate files by copying entire directories. Once the copying is identified we propose an algorithm to link version histories from multiple repositories in order to construct universal version history. The results show that about 41.32% of source files (in the repository involving more than 6M versions of around 2M files) were duplicated among the Avaya's source code repositories for more than ten different projects. The prefix-postfix algorithm is more suitable than prefix algorithm due to the reasonable error rates after validation of the known copying behaviors.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance and Enhancement – Restructuring, reverse engineering, and reengineering

General Terms: Algorithms, Measurement

Keywords
Cloning, Version Control, Change History, Code copying, Code Authorship

1. INTRODUCTION
Software reuse is always a key factor to increase the speed of the software development. Studies done by Baker (1995) and Lague (1997) suggested that duplicated codes were 5% to 10% of the source codes of large programs. Most projects are implemented based on an existing piece of codes. Therefore, it is generally believe that code cloning or code coping is often used in developing industrial systems. Companies may use different version control systems to maintain different projects due to the evolution of version control systems and cross-organization work. Hence, cloned codes are often distributed among different version control systems.

A previous study [3] proposed that cloned sources were more reliable than non-clone code. In that particular study the maintenance cost of a duplicated module was less than for a non-duplicated module. Furthermore, knowing clones may help fix bugs in all cloned copies if a bug is detected in any one of them. Understanding how code is cloned allows us to identify the original authors of the program as well as determine the code sharing relationships and interdependencies between people and projects over time.

There are various kinds of cloned code detection methods that have been proposed. Baxter (1998) used abstract syntax trees to find out the cloned codes. Ducasse (1999) proposed a pattern matching technique that divided programs into strings and then the duplications were caught by comparing those strings against each other. Kamiya (2002) developed a language independent tool - CCFinder to find clone codes by matching token sequences that were transformed from source files by lexical analysis. In addition, in order to overcome the navigation difficulties of a huge set of results generated by clone detection tool, Kapser (2005) implemented a supportive tool for visualizing the clone codes detected from CCFinder. However, these methods might not be suitable to detect clones for source code version history repositories because they could take enormous computation efforts even on a single version of the code. With more than six million versions in the repository we have analyzed assuming average file size of 500 lines the CCFinder would require 13 days if computer memory is not exhausted. We base this estimate on the published 3 minute execution time for 500K lines of code assuming linear complexity for CCFinder. Furthermore, the clone size those techniques are targeting is fairly small in comparison to a large software project. Moreover, the copying behaviors of various versions of one project or projects to projects have not been discussed in the existing methods. Therefore, we look at algorithms that are appropriate for large code repositories involving version histories.

The purpose of this paper is to introduce two algorithms that were applied to detect the duplicated files in Avaya's version control repositories. A universal version history over different version control systems was built according to the identified copies of the files.

MSR'06, May 22–23, 2006, Shanghai, China.

2. METHOD

2.1 Methodology

It is computationally infeasible to detect cloned codes at the function or syntax level for a very large population of files. Therefore, our approach was to extract the possible clone code files by examining the full pathnames of source code files in the initial stage of clone detection and then find the duplicate code from the candidate files. This allowed us to avoid computationally intensive step of comparing the content of the files.

Base on the observations of product development we identified a number of scenarios that could cause identical files with different pathnames.

(1) A substantial amount of code copying takes place when a software project copies the code base for a new release or for a similar but distinct product.
(2) While such copies can be realized as branches in a version control system, some projects create separate instances of the version controlled files by copying them into a separate directory.
(3) A different version control repository is usually used if another organization is developing a similar product and wants to have their own version control repository for the code.
(4) In some cases, projects change the version control tools, forcing the change in repositories.

Our definition of duplicated source files is that files have at least one nonempty version between them (excluding spaces). In other words, let a_1, …, a_n be versions of file a and b_1, ..., b_m be versions of file b. Then a and b are duplicate if exists 1<=i<=n and 1<=j<=m, such that a_i is identical to b_j (excluding white spaces) and a_i is non-empty.

The methodology of this paper (see Figure 1) contains five primary steps. Firstly, a list of filenames was analyzed and then two algorithms were used to detect the common files. According to known copies, the algorithms were calibrated or validated. To trace authorship and version history of cloned files, a universal version history was finally created by connecting the histories of individual cloned source files.

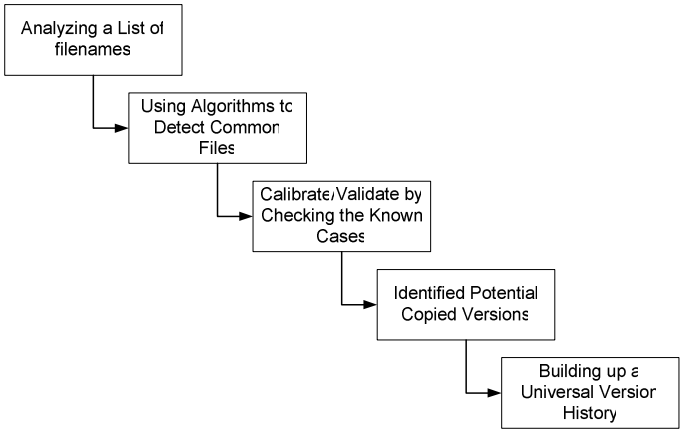


Figure 1. Approach procedure

2.2 Algorithms

Based on the observed developers' copying behaviors, in order to duplicate files, developers often copy the entire directory. Hence, the files sharing a name in any two folders may be identical. Therefore, the directories that share a large fraction of filenames should be identified first and then cloned files can be obtained by comparing the files with a common name in such directories. As a result, the goal of the following two algorithms is to retrieve the potentially copied directories.

2.2.1 Terminology

To simplify presentation we introduce some terminology first.

(1) Division level: the full file path can be broken into levels according to the '/' symbol (the repositories we investigated are mostly in UNIX file system. We have converted Windows '\' directory separator into UNIX '/' for the few version repositories kept in Windows file systems).
(2) Prefix and postfix: in the whole directory path of a file, the part of the path preceeding the division level is prefix and the remaining part of the path is postfix. For example, for a path D_i/D_j/D_k. If D_i is the prefix, D_j/D_k is the postfix if the division level is one.
(3) Common directory pair: if two directories are thought to be the copied, these two directories are called the common directory pair
(4) Common prefix pair: if two prefixes are thought to be the copied, these two prefixed are called the common prefix pair.

2.2.2 Prefix algorithm

The assumption of prefix algorithm is that directories i and j are considered as candidates for common directory pairs as long as there are at least certain portion of identical filenames in i and j. If Ni and Nj are the number of files in i and j as well as n_{ij} is the number of identical files in i and j, we use the following criteria in order to identify common directory pairs among different version repositories. The n_{ij} divided by N_{min} should be larger than a cutoff coefficient, where N_{min}= min(Ni, Nj). In other words, the fraction of files with identical names has to exceed a cutoff for a smaller directory in order to consider two directories to be potentially copied. We use this criterion to make sure that common filenames used in many unrelated projects such as, system.h, config.h, and main.c, do not affect our algorithm.

As long as those common directory pairs were extracted, a group of identical directories can be produced by using transitive law; that is, if we found a common directory pair (i,j) and a common directory pair (j,k), directories i, j, and k are identical. After a group of identical directories was gathered, those potential identical files can be matched under those identical directories.

2.2.3 Prefix - Postfix algorithm

The prefix – postfix algorithm is very similar to the prefix algorithm with one additional procedure. The new assumption is that prefixes i and j are considered as candidates for the common prefix pair if the result of the number of common postfixes following prefixes i and j divided by the minimum of the number of postfixes after prefixes i and j is larger than a cutoff coefficient of the criterion, the prefixes are considered to be common. Instead

of counting the fraction of common files in two directories as in the prefix algorithm, we are counting the fraction of postfixes.

Like gathering identical directories in prefix algorithm, those identical prefixes can be grouped and then the identical postfixes under the identical prefixes can be found.

2.3 Creating the Universal Version History

The files that are identified as copied are ordered according to the initial creation time of each file. The oldest one of them is assumed to be the main trunk of the universal version, while the other files are connected to the main trunk. If two files do not share the content for any version, they are considered not copied.

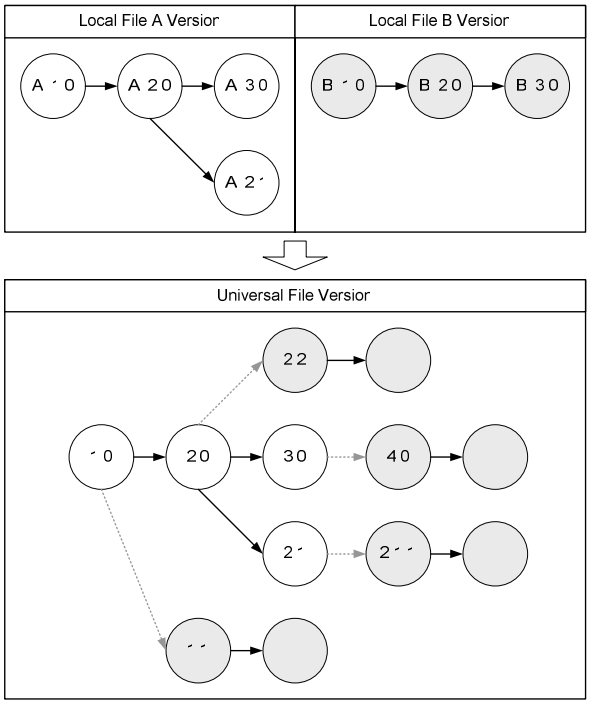


Figure 2. Universal version building procedure

For example, if two files, A and B are copied (see Figure 2), four possible merges of version histories may be appropriate depending of the versions of file A and File B that are identical. The assumption of this rebuilding process is that the first version of the later file should be copied from some version of the earlier file and then the later file is changed. Therefore, this approach has been simplified. We only connected the first version of the branch file to the main trunk file even though a later version of branch file was the same as any versions of the main trunk file. For instance, we do not connect 2.0 and 3.0 of B to any versions of A (see Figure 2).

3. RESULTS

The Avaya's version repositories we have analyzed contained more than 6M versions of more than two million source code files. From our previous work with various projects, we have identified the known cloned cases. We used the Type I & II error method to validate both algorithms. Type I error identifies the fraction of file pairs (i, j) not known to be copied that were identified by our algorithms as copied. Similarly, Type II error is the fraction of the pairs (i, j) known to represent copied files that our algorithm did not identify as copied. The population size of directory pairs is n(n-1)/2, where n is the number of the total directories we explored. Obviously, the number of d_i = d_j represents only a small part of the population; therefore, the Type I error for both algorithms were small. However, the Type II error of the prefix algorithm was fairly large (see Table 2, 3). Therefore we used prefix–postfix algorithm to identify the copied files.

Table 1. Type I and II errors of prefix – postfix algorithm

		Experimental results	
		d_i <> d_j	d_i = d_j
Known cases:	D_i <> D_j	2154385410	Type I error: 28804 (Rate: 1.34E-03%)
	D_i = D_j	Type II error: 822 (Rate: 1.5%)	53867

Table 2. Type I and II errors of prefix algorithm

		Experimental results	
		d_i <> d_j	d_i = d_j
Known cases:	D_i <> D_j	2153498853	Type I error: 915361 (Rate: 0.042%)
	D_i = D_j	Type II error: 20871 (Rate: 38.1%)	33818

An observation of the program file naming behavior indicates some common filenames that are often used in many unrelated software projects; for example, programmers often used “main.c” for the entry program of C or C++, “help.c” for the instruction file or “main.h” for include files. Hence, we excluded these high frequency filenames when counting the number of identical filenames (n_{ij}) between two directories. The experiment showed that about 41.32% of the total files were identical. These common files came from different projects and programming languages.

The algorithms are implemented in the Perl programming language and were run on a Sun V40Z machine that contains 16G RAM and dual Opteron CPUs, running SunOS 5.10. for processing more than 6M versions of more than 2M files. (Each file had its version history.) The approximate computational time of these two algorithms and the rebuilding procedure is shown in Table 3.

Table 3. Computation time comparison

		Time
Clones Extraction	**Prefix**	2.1 hours
	Prefix – postfix	5.1 hours
Universal Version Construction		10.2 hours

The whole universal building process includes two steps – (1) matching the identical files from the common directories that we

acquired via prefix-postfix algorithm; (2) integrating versions among the identical files. The total time of this process is about 10.2 hours.

4. DISCUSSION

We presented two practical algorithms for detecting the copied files in multiple large version control repositories. Although the prefix – postfix algorithm could provide more accurate results, it needed much more computation time than the prefix algorithm. These two detection methods identify copies only if filename does not change. Therefore, we cannot detect the copies if the programmers modified the filename. Besides, these methods may not be effective for other copying strategies that are not considered in our assumption, like programmers randomly copy the file without duplicating the entire directory. However, there are some advantages of this approach. It can identify clones over multiple versions, for various programming languages, and for multiple version control repositories. The processing speed of this method is also acceptable for identifying clones in multiple very large version repositories. The main value of constructing universal version history is to identifying code authorship and copying patterns over multiple large repositories.

5. FUTURE WORK

Although prefix – postfix algorithm provides some advantages for finding the clones of different programming languages in distributed huge repositories, the method can only identify identical source files; that means, similar but not identical files or copied source code segments cannot be found. This implies that those more detail level clone identifying methods are complimentary to our approach and could be combined in future studies.

Since we did not consider more complicated possible version history construction situations, a more complete universal version history building method should be further investigated.

The scripts used to implement the described algorithms are available upon request from authors.

6. ACKNOWLEDGMENTS

Our thanks to Avaya Labs Research for providing us all the resources.

7. REFERENCES

[1] Brenda Baker. On finding duplication and near duplication in large software system, IEEE Working Conference on Reverse Engineering 1995.

[2] B. Lague, D. Proulx, E. Merlo, J. Maryland, J. Hudepohl, Assessing the benefits of incorporating function clone detection in a development process, IEEE International Conference on Software Maintenance 1997.

[3] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software, Proceedings of the 8th International Symposium on Software Metrics 2002.

[4] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo SantAnna and Lorraine Bier. Clone detection using abstract syntax trees. In Proceedings of the 8th International Symposium on Software Metrics 1998.

[5] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. International Conference on Software Maintenance 1999.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Engineering, Vol. 28, No.7, 2002.

[7] Cory Kapser and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. International Conference on Software Maintenance 2005.

Concern Based Mining of Heterogeneous Software Repositories

Imed Hammouda
Tampere University of Technology
Institute of Software Systems
P.O. Box 553
FI 33101 Tampere, Finland
imed.hammouda@tut.

Kai Koskimies
Tampere University of Technology
Institute of Software Systems
P.O. Box 553
FI 33101 Tampere, Finland
kai.koskimies@tut.

ABSTRACT

In the current trend of software engineering, software systems are viewed as clusters of overlapping structures representing various concerns, covering heterogeneous artifacts like models, code, resource files etc. In those cases, adequate search mechanisms for software repositories should be based on such fragmented nature of software systems, allowing concern-oriented queries on the system data. For this purpose, we propose a conceptual framework for a concern-oriented query language for software repositories. A pattern-based implementation scheme is discussed, exploiting existing tools. The applicability of the approach is studied in the context of an industrial case study.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Miscellaneous

General Terms

Documentation

Keywords

Heterogeneous software repositories, concern-based mining, pattern-based search structures

1. INTRODUCTION

Software repositories have traditionally been understood as collections of source artifacts that consist of either monolithic source text [15] or higher-level data structures storing the essential information contained by the source [5]. In both cases, the information in the repository is organized according to the structures and semantics of the underlying programming language. Using this kind of organization creates a gap between the repository and the needs of various stakeholders: it is often difficult to express high-level information needs in terms of the low-level concepts provided by the repository.

For example, a maintainer may be interested to identify those parts of the system that are related to the persistency issues concerning the copy-and-paste feature of the system. Mapping such information request to the source structures is virtually impossible without substantial additional knowledge about the ways copy-and-paste and persistency are implemented in the system, no matter how detailed information about the source is stored in the repository.

We argue that there should be a convenient mechanism that allows us to incorporate information about the anticipated concerns of the system stakeholders into the software repository, and that the search engine should exploit this information. In this way the high-level requests of the stakeholders can be mapped to the low-level structures of the source. The required information is inherently external: it cannot be inferred from the system with any automatic means. This is in contrast to traditional techniques such as adding special comments to code or informal notes to UML diagrams [17]. However, the results of queries may introduce new information which can be cumulated in the repository.

Another major requirement for a software repository is its ability to support heterogeneous software artifacts. Many stakeholders are not interested in source code only, but in requirements and feature models, in architectural and design models, in scripts, resource files etc. Most concerns are related to several artifact types expressed in different languages and notations. Thus, queries on the software repository should yield results that span heterogeneous artifacts.

In this paper, we propose a concern-based organization of a software repository, with an external structure representing the relevant concerns. The result of a query is always a concern as well, possibly added to the set of persistent concerns. The actual software artifacts are not touched, but only referenced in the external structure. The overall approach is depicted in Figure 1.

The technical solution for representing concern structures may vary. In this paper, we investigate the possibility to use a pattern-based approach, which has certain advantages. In particular, this approach lends itself to capturing software elements in heterogeneous artifacts, fast concern manipulation, overlapping concerns, and existing tool support. However, the basic idea could be realized using other techniques, too.

The resulting concern-based information about the software system can be displayed and exploited in various ways.

MSR'06, May 22–23, 2006, Shanghai, China.

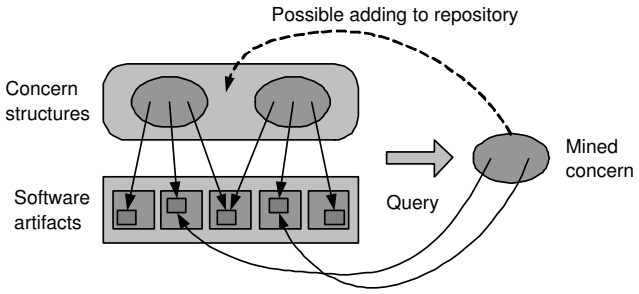


Figure 1: Concern-oriented repository mining.

We assume that the concerns (including the concerns resulting from the queries) can be traced and visualized by a tool, allowing the construction of various concern-based views on the system.

We will proceed as follows. In the next section we discuss the multidimensional nature of software systems and the use of concerns as the basic unit of queries. In Section 3 we briefly explain the concept of a pattern and how it can be used to represent concerns in a software repository. Section 4 briefly discusses tool support for the pattern-based mechanism. The approach is demonstrated in the case of an industrial system in Section 5, and some concluding remarks are presented Section 6.

2. DECOMPOSITION OF SOFTWARE AR TIFACTS

We assume that the artifacts of a software system consist of sets of artifact elements. The elements of a set have the same type, e.g model or code elements, but the sets are heterogeneous. Each set is associated with a specific artifact type and addresses certain concerns in the software system. From the concern point of view, a set can address one or many concerns. For instance, given an MVC-based (Model View Controller [13]) implementation of an arbitrary software system, the code representation of the architectural style may address several functional features of the system. Thus, from the architectural point of view the MVC part of the system addresses one concern, but from the functional point of view several concerns are involved.

This decomposition scheme is discussed in Figure 2. The top cubical structure illustrates the decomposition of a system based on its artifact types. Each top cube represents a set that groups together artifact elements that are of the same type. Each of these sets can be further decomposed into smaller sets based on the concerns addressed in the artifact elements of that set.

The fragmented nature of software systems is naturally reflected in the process of mining software repositories as the goal of any repository search operation is to retrieve the elements of that repository that address a certain matter of interest. Furthermore, the search operations themselves could be based on the outcome and the combination of other search operations. To give an example, system maintainers are usually interested in those parts of the system data that are relevant to their actual maintenance needs. As the context of maintenance changes during the system evolution process, new fragments of the system, in addition to the current ones, might become relevant. The natural relationship between system fragments and the concerns they treat advocates the need for a concern-oriented query language. As the fragments represent heterogeneous sets of repository elements, we can express the elements and the operations of this language in terms of set theory [7] concepts and terminology.

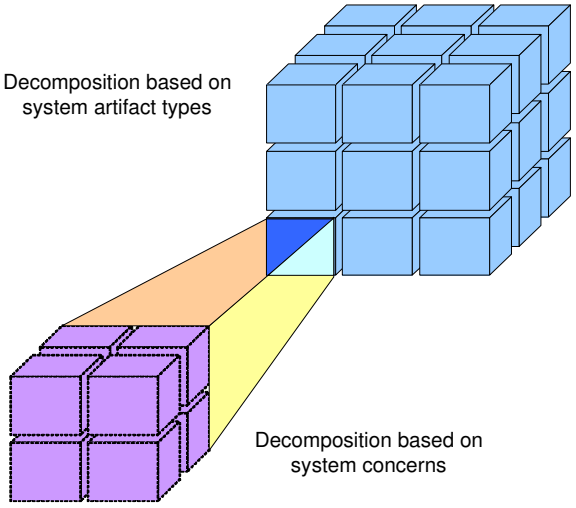


Figure 2: Decomposition of a software system.

In the context of concern-based decomposition of software repositories, we can define a set (S) as an unordered group of artifact elements (E0, E1, E2 ...) with no duplicates. It is possible, however, to decompose sets of repository elements into smaller sets (S = S0 + S1 + S2 ...). Considering the example given earlier, the MVC-based source code can be decomposed into smaller sets of code elements, each representing a certain functional feature. During the decomposition process, the same artifact element may be placed into different sets. This is the case for elements that address multiple concerns.

There are two special cases of sets: the empty ($\emptyset$) and the universal set (U). For software systems, the universal set is a heterogeneous collection of elements that represent all system artifacts. No concern-based logical system decomposition, however, can lead to sets that are empty. Empty sets can only be results of arbitrary search operations performed on the concern-based decomposition of the system.

Based on the above discussion, let us consider two arbitrary system concerns C1 and C2 and two sets S1 and S2 containing the artifact elements addressing concerns C1 and C2, respectively. We can define different concern-based queries on C1 and C2. We assume that sensible concern-based queries can be formulated using the classical set operations: union, intersection, complement, and difference. While other set operations (like Cartesian product) could be realizable as well, it seems that they are less useful in practice for concern-based queries.

- *Concern merging.* The merging of the two concerns C1 and C2 can be expressed as the union of the two sets S1 and S2, which is the set S containing all the elements in either S1 or S2. The elements of S address either concern C1 or concern C2. This operation is denoted using the '+' symbol: C = C1 + C2.

- *Concern overlapping.* The overlapping of the two concerns C1 and C2 can be expressed as the intersection of the two sets S1 and S2, which is the set S containing all the elements that are in both S1 and S2. The elements of S address both concerns C1 and C2. This operation is denoted using the '&' symbol: $C = C1 \ \& \ C2$.

- *Concern slicing.* The slicing of the concern C1 with respect to the concern C2 can be expressed as the difference of the two sets S1 and S2, which is the set S containing all the elements that are in S1 but not in S2. The elements of S address concern C1 but not concern C2. This operation is denoted using the '-' symbol: C = C1 - C2.

- *Concern exclusion.* The exclusion of concern C1 can expressed as the complement of the set S1, which is the set S containing all the elements in the universal set U except those in S1. The elements of S address all other concerns except concern C1. This operation is denoted using the '^{c}' symbol: $C = C1^c$.

To give an example, consider a software system with a repository containing models, programs, resource files, project documentation, etc. The software system provides different security strategies and requires user authentication for many of the functionality it implements. We can, therefore, identify two system concerns: security (say Sec) and user authentication (say Auth). In the context of those two system concerns, the four above concern operations can be viewed as searching for all repository elements addressing security or user authentication (merging, Sec + Auth), elements representing user authentication security (overlapping, Sec & Auth), elements corresponding to all security aspects except those for user authentication (slicing, Sec - Auth), and elements addressing all other concerns except security strategies (exclusion, Sec^c).

3. PATTERNS AS SEARCH STRUCTURES

As a tool infrastructure for concern-based querying, we use the concept of aspectual patterns [10] to represent arbitrary concerns in software repositories. Figure 3 depicts a conceptual model in UML for aspectual patterns. A pattern is a collection of hierarchically organized roles rather than concrete repository elements. A pattern is used to collect together all repository elements that address a certain system concern. This is done by binding pattern roles to certain elements of the repository representing that concern. Each role can be associated with a set of properties that can be used for the search operations. An example property could be a textual description tag explaining the purpose of the role binding. Another example property is the type of the role. The role type determines the kind of repository elements that are bound to the role, e.g. model elements (UML elements), source code (Java elements), resource data (binary documents), project documentation (text file and file fragments), etc.

A concern-based decomposition of a software repository partitions that repository into a number of fragments. At the elementary level, each fragment corresponds to exactly one system concern. Each of these fragments is represented by a separate pattern.

The elements of a fragment may refer to or depend on each others. In many cases, the relationship between two

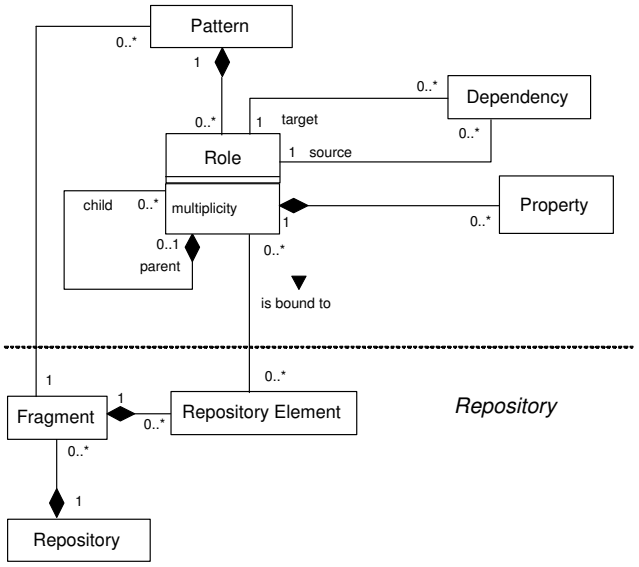


Figure 3: Conceptual model for aspectual patterns.

elements is implicit and cannot be marked in the fragment. For example, it might be hard to express that a binary file is a resource for a certain implementation class. Such a relationship can be easily made explicit in the pattern using role dependencies.

It is possible to bind the same pattern role to multiple repository elements. This is the case if multiple elements play that specific role. The possibility to bind a role to multiple elements (and the optionality of certain roles) is indicated using the multiplicity attribute of the role. For example, if a role has multiplicity [0..1], the role is optional in the pattern (and thus in the concern).

Figure 4 illustrates how patterns are bound to repository elements. There are two patterns X and Y. Each pattern represents a separate system concern and is associated with a repository fragment. Each fragment contains the elements that address the corresponding concern. The fragments do not have to be aligned with the physical or logical distribution of the repository. In Figure 4, the fragment that is associated with pattern Y contains elements cutting across different packages. Furthermore, the example fragments are overlapping, there is a repository element that addresses both concerns. Roles r2 and r3 are bound to the same element while r7 represents the role of two different elements. The repository may contain elements that are not referred by the patterns. Such elements do not correspond to any anticipated concern and do not manifest in the repository mining process. Similarly, patterns may have unbound roles. Such roles may represent possible extension points of the corresponding fragments.

Using role-based pattern structures, the concern operations we identified in the previous section can be realized in a straightforward way.

- *Merging operation.* The merging of the two patterns X and Y returns a new pattern with roles bound to all repository elements referred by either X or Y. In the example case, the resulting pattern references elements of both fragments (six elements).

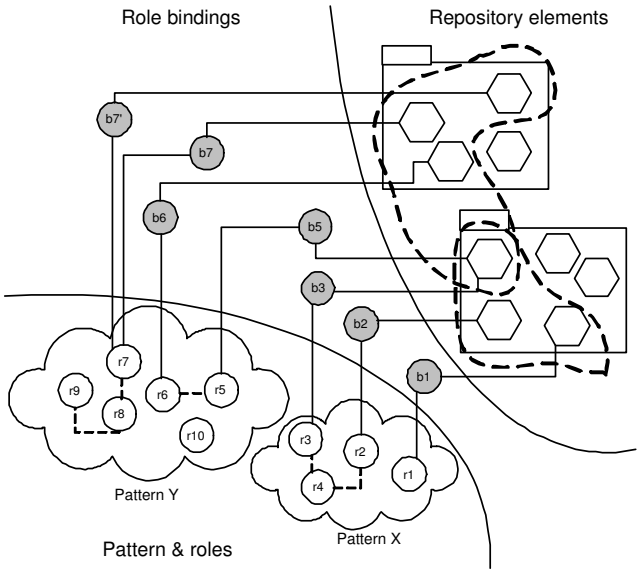


Figure 4: Aspectual patterns as search structures.

- *Overlapping operation.* The overlapping of the two patterns X and Y returns a new pattern with roles bound to all repository elements referenced by both patterns. In the example case, the resulting pattern references one element (corresponding to binding b3 and b5).
- *Slicing operation.* The slicing of pattern Y with respect to pattern X returns a new pattern with roles referring to repository elements bound to roles of Y but not to roles of X. In the example case, the resulting pattern references three elements (corresponding to bindings b6, b7 and b7').
- *Exclusion operation.* The exclusion of pattern Y returns a new pattern referring to all elements not bound to the roles of Y. In the example case, the resulting pattern references two elements (corresponding to bindings b1 and b2). Note that the universal set represents only the bound repository elements, the other unbound elements (three elements) are not considered for any concern queries.

In addition to the above search mechanisms, patterns can be exploited to provide support for other ways of mining tasks. This is achieved by using the properties attached to pattern roles or by accessing the properties of the referenced repository elements themselves. For instance, we might be interested in retrieving repository elements of certain types or simply those elements whose role properties match certain criteria.

As explained earlier, the approach discussed in this paper assumes that the repository comes with an initial annotation that represents an original concern-based clustering of the repository elements. Each cluster addresses a specific system concern and is therefore represented by a separate pattern. During the mining process, the repository is explored by visiting the appropriate patterns, following role bindings (to the repository elements), and investigating the properties of the pattern roles. Based on the original patterns, new clusters corresponding to other concerns (and concern combinations) can be constructed by applying the search mechanisms discussed earlier. As the search results are in fact returned as new patterns, these new patterns could be added to the original annotation and might themselves be used in future mining operations. In the case study section, we will give a concrete example of such a scenario.

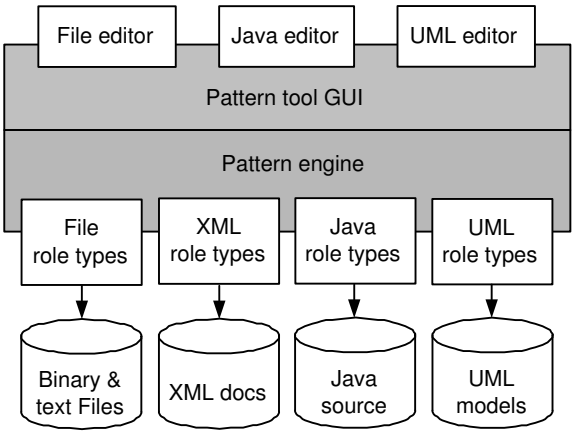


Figure 5: Architecture of the MADE environment.

4. TOOL SUPPORT MADE

In order to demonstrate the pattern-based approach for representing concern structures, we use an Eclipse-based [6] pattern-driven development environment called MADE (Modeling and Architecting Development Environment [11]). Figure 5 depicts a layered architecture of the MADE tool environment. The pattern engine represents the core component of the platform. It is used to manage the binding process and is thus independent of any artifact types. Currently, the environment provides support for binding pattern roles to UML, Java, XML, and general file (binary and text) elements.

The pattern tool GUI provides different views and wizards for creating new patterns, adding roles to patterns, binding roles to repository elements, viewing role bindings, and tracing bindings to their corresponding repository elements. When a bound element is retrieved, it can be viewed in its own editor. Currently Rational Rose is used as the UML editor. Therefore, only Rose-based UML models can be browsed. For Java, XML, and general file content, Eclipse-built-in editors are used.

Furthermore, the MADE platform comes with mechanisms for browsing and highlighting patterns (concerns) in the repository data. The biggest limitation of the tool, however, is that search operations, as presented in this paper, are not currently implemented. We are planning to add such support in future releases of the platform. Nevertheless, it is still possible and beneficial to use the tool in its current state to construct an original concern-based annotation of a repository and present that as a documentation tool for system learners. Another limitation of the tool is that MADE pattern roles are strongly typed. In order to support other repository element kinds (such as other programming language elements), the tool has to be extended with new role types.

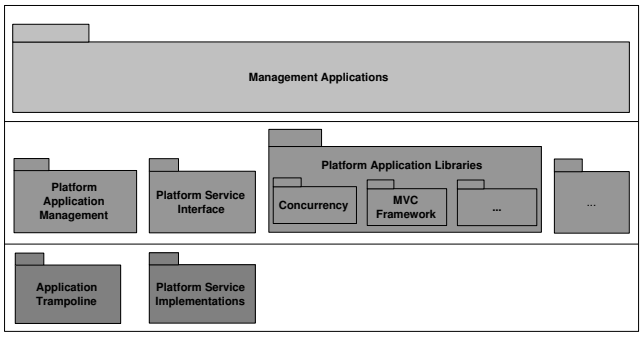


Figure 6: Architecture of Nokia GUI platform.

5. CASE STUDY

Network management represents one of the core businesses of Nokia. For managing networks and network elements, the company produces a family of NMS (Network Management System) and EM (Element Manager) applications. The GUI parts of the applications are developed based on a common platform. The main purpose of the platform is to help developers to build Java-based GUIs and to make sure that NMS and EM applications share the same features. The GUI platform has been developed as an object-oriented Java framework. New management applications are constructed by specializing parts of the framework and using the common services offered by the platform.

Figure 6 depicts a logical decomposition of part of the GUI platform consisting of several independent logical blocks. The upper layer (Management Applications) stands for the NMS and EM applications. They are not part of the platform but they are built on top of it. The middle layer shows a number of platform subsystems:

- Platform Application Management. The purpose of this subsystem is to keep track of running applications and to manage the starting of new applications.
- Platform Service Interface. This component defines the abstract interfaces to the services offered by the GUI platform. Example services include logging, authentication, and online help facilities
- Platform Application Libraries. This subsystem provides common features used for building new network management applications. Example features include concurrency control and an MVC-based framework

The bottom layer shows a block representing Platform Service Implementations. The GUI platform comes with a number of default service implementations. If needed, application developers can provide their own service implementations based on the service interfaces. The Application Trampoline component represents an external interface to Application Management for allowing processes outside the virtual machine to start new applications in the same virtual machine.

The system repository for the platform comes with a wide range of artifact types including design models, source code, executable jars, property files, deployment descriptors, user manuals, technical development documents, managerial presentations, etc. In addition to platform artifacts, the repository also contains example applications built on top of the platform.

Table 1: Example platform concerns

Concern Category	Concerns	Artifact types
Component concerns	Application Management (AppMan), Application Trampoline (AppTram)	Jars, UML design models, Java source files, documents
Feature concerns	Authentication (Auth), Online help (OnlineHelp)	Jars, UML feature and design models, Java source files, XML descriptors, documents
Architectural concerns	MVC (MVC), Layered architecture (layered)	UML architectural and design models, Java source files, documents
Maintenance concerns	Adding alternative service implementation (AltServImp), Modifying service interfaces (ModServInt)	UML design models, Java source code, documents
Specialization concerns	Feature specialization (FeatSpec), GUI specialization (GUISpec)	UML design models, Java source code, XML descriptors, documents
Global concerns	Persistency (Pers), Security (Sec)	UML design models, Java source code, documents

As an original annotation of the repository, we have identified a number of system concerns of different categories. By concern categories, we mean groups of related system concerns representing similar matters of stakeholders' interests. Table 1 depicts six concern categories. For each concern category, we give two example concerns. For instance, from the viewpoint of system components, one can identify a concern standing for application management and another representing application trampoline. Each of these system concerns is represented using a separate aspectual pattern. The name of the pattern is given next to the concern definition. In addition, the table gives the artifact types where each concern is represented.

In order to illustrate our concern-based query language, Table 2 shows four example concern queries based on the concerns (patterns) identified in Table 1. The first query stands for concern slicing. It is for excluding all repository elements that address the authentication concern (Auth) and are at the same time related to system security (Sec). The search results of this query are themselves returned as a pattern (NonSecAuth). The latter pattern is then used in the second query to narrow down the search results further to those elements that are also addressing application trampoline. The third query gives an example of a concern merging operation and the fourth shows a second illustration of the merging operation.

Table 2: Example concern queries

Stakeholder interest	Concern expression
Are there any parts in authentication that are not related to security?	NonSecAuth = Auth - Sec
Are any of the parts above in the application trampoline component?	NonSecAuth & AppTram
Show me the application management component in the context of the MVC architecture!	MVC + AppMan
What are the parts of the platform that are relevant for creating online help for an application?	FeatSpec & OnlineHelp

In some situations, the user might not be interested in viewing all the artifact types corresponding to her concern query. For instance, designers may prefer to analyze system repositories based on detailed design diagrams. In the context of tool support, it is useful to allow users to refine the query results based on their desired artifact types.

6. DISCUSSION

In this paper, we have argued that in the current trend of software engineering, mining mechanisms are driven by the heterogeneous and fragmented nature of software systems. Such a viewpoint has already been taken in several research works in the field of mining software repositories [4, 16]. Supporting such a viewpoint, we have presented a conceptual framework for a concern-oriented query language. Our approach differs from other querying models, such as the one presented in [12], by treating system concerns as first class citizens. The concerns, grouped according to various concern categories, address different matters of stakeholders' interests.

Concern-oriented mining has been widely discussed in the field of aspect-oriented software development (AOSD [9]) and is generally referred to as concern elaboration. There are many tools that can be used for concern elaboration. These include standalone search tools, browsers integrated in IDEs, and compilers. A typical scenario, using these tools, is to run a query over a model to retrieve the model elements addressing a certain concern. Similar to our approach, the queries could be refined once the results are evaluated.

In [14], three concern elaboration tools have been evaluated namely, AspectBrowser [1], AMT [2], and FEAT [8]. These tools differ in two major ways: how programs models are represented before being elaborated and how search results are presented to the user. Instead of relying solely on repository data for extracting the required information, our approach is based on mining a search space that is external to the repository. In our methodology, the search space is not deduced from the repository but rather superimposed on the repository data. This is in contrast to other approaches (i.e. reverse engineering [3]) that are based on building high level representations of low level system data [5]. It can be argued that the two approaches are in fact complementary.

As an implementation scheme, we have used a role-based pattern mechanism for representing system concerns and formulating the queries. Using patterns we could successfully annotate a repository taken from the industry and we could conveniently express our search queries. As an experimental environment for concern-based mining, we have exploited an existing pattern-driven development environment known as MADE. The MADE environment comes with various capabilities for concern-based annotation of software repositories. However, search mechanisms need still to be implemented in the tool.

7. ACKNOWLEDGMENTS

This research has been financially supported by the National Technology Agency of Finland (project Inari), Nokia, Plenware Group, TietoEnator, and John Deere.

8. REFERENCES

[1] Aspect Browser WWW site. Available at `http://www-cse.ucsd.edu/users/wgg/Software/AB/`, 2006.

[2] Aspect Mining Tool (AMT) WWW site. Available at `http://www.cs.ubc.ca/~jan/amt/`, 2006.

[3] E. Chikofsky and J. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[4] A. Dekhtyar, J. H. Hayes, and T. Menzies. Text is software too. In *Proc. MSR 2004*, pages 22–26, Edinburgh, Scotland, UK, 2004.

[5] S. Demeyer, S. Tichelaar, and S. Ducasse. Famix 2.1 - the famoos information exchange model. Technical report, University of Berne, 2001.

[6] Eclipse WWW site. Available at `http://www.eclipse.org`, 2006.

[7] H. B. Enderton. *Elements of Set Theory*. Academic Press, 1977.

[8] Feature Exploration Tool (FEAT) WWW site. Available at `http://www.cs.ubc.ca/labs/spl/projects/feat/`, 2006.

[9] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[10] I. Hammouda. A tool infrastructure for model-driven development using aspectual patterns. In S. Beydeda, M. Book, and V. Gruhn, editors, *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, pages 139–178. Springer, 2005.

[11] I. Hammouda, J. Koskinen, M. Pussinen, M. Katara, and T. Mikkonen. Adaptable concern-based framework specialization in UML. In *Proc. ASE 2004*, pages 78–87, Linz, Austria, 2004.

[12] A. Hindle and D. M. German. SCQL: a formal model and a query language for source control repositories. In *Proc. MSR 2005*, pages 100–104, Saint Louis, Missouri, USA, 2005.

[13] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in the

smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[14] G. Murphy, W. Griswold, M. Robillard, J. Hannemann, and W. Wesley Leong. Design recommendations for concern elaboration tools. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 507–530. Addison-Wesley, 2004.

[15] Rational ClearCase WWW site. Available at `http://www-306.ibm.com/software/awdtools/clearcase/`, 2006.

[16] G. Robles and J. M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. In *Proc. MSR 2005*, pages 106–110, Saint Louis, Missouri, USA, 2005.

[17] Unified Modeling Language WWW site. Available at `http://www.uml.org/`, 2006.

Software Engineering Applications of Logic File System

Application to Automated Multi-Criteria Indexation of Software Components

Benjamin Sigonneau
IRISA/Université de Rennes 1
Rennes, France
benjamin.sigonneau@irisa.fr

Olivier Ridoux
IRISA/Université de Rennes 1
Rennes, France
ridoux@irisa.fr

ABSTRACT

Logic information systems use formal concept analysis in a novel way to manage information. A file system implementation has been designed under the name of Logic file system. It offers a flexible management of non-hierarchical data. We present several applications of Logic file system to software engineering: multi-criteria indexation of software components, multi-concern browsing of source files, and bug finding in test traces.

We detail multi-criteria indexing of software components. Three independent indexing frameworks are developed and merged in a single multi-criteria framework. The three indexing frameworks capture formal criteria like type isomorphisms and inheritance relations, semi-formal criteria like naming conventions, and informal criteria like keywords of comments. We show how the logical orientation of Logic file system helps in capturing all these criteria in a single framework.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environment

General Terms

Design

Keywords

Logic information system, Software components

1. INTRODUCTION

Software engineering manages many kinds of documents that are related in many ways; it is the place for cross-cutting concerns and multi-view schemata. However, this rich network of relationships is often flattened on a single hierarchy. This is sometimes called the "tyranny of the dominant decomposition" [19]:

- The Java class browser is organized after the inheritance hierarchy; this makes it nearly impossible to search a method using different criteria (e.g. search for a method that returns a string, or search for a method that raises a particular exception).
- A standard source file displays the linear hierarchy of a text. One cannot claim to organize a source file using several criteria at the same time. One criterium must dominate the others.
- Object orientation brings types to the top of the hierarchy, so examining or creating a type is easy. At the opposite extreme, procedure orientation brings routines to the top of the hierarchy, so that examining or creating what concerns a type is scattered across source files, but examining or creating routines is easy.
- Sometimes, a hierarchy is imposed by a programming language. See for instance the layout of Java packages on directories.
- Even when it is electronic and enriched with navigation links, documentation is often thought as a rigid document, in which an almost arbitrary priority of concerns concentrates favoured concerns in a few sections, and discards unfavoured concerns across the whole document.

In all these situations, finding a particular piece of information requires either luck or erudition. We contend that neither luck nor erudition is a necessary practice for robust software engineering.

We feel that the very heart of this problem lies in the fact that traditionnal information systems (e.g. hierachies or databases) fail to organize this data in an efficient way that would be convenient to the user, and artificially enforce the use of a dominant decomposition. We propose to use a new powerful organization framework, called *Logic Information System* (LIS), to manage software engineering artifacts.

In Section 2 we will present what a logic information system is and a file system implementation of a LIS. Then we will present in Section 3 the outlines of three experiments of software engineering applications based on the LIS file system. Finally, we will present in more details the retrieval of Java methods using the LIS file system (see Section 4).

2. LOGIC INFORMATION SYSTEMS

Hierarchical organizations are rigid because only one path leads to every single object. Sometimes, this rigidity is relaxed by introducing links, but this does not scale well because navigation does not work well with links, and generally nothing prevents from creating dangling links. In short, links are an afterthought[1]. The good news about hierarchical organization is that it makes a notion of place very intuitive (every node in the hierarchy is a place), and it

[1]Even in cases where links are not an afterthought, e.g. in a web site, the same rigidity arises since only a *fixed* set of paths leads to every object.

MSR'06, May 22–23, 2006, Shanghai, China.

makes navigation progressive. Indeed, if we consider the hierarchy of a UNIX file system, the answer to an `ls` in a place (e.g. a directory) is the set of objects that inhabit the place, and a set of sub-places (e.g. subdirectories). It is only when forced that a hierarchical organization dumps all its content (say, using `ls -R` in a UNIX file system).

A truly different organization is the boolean organization, where objects are associated with attributes, and the answer to a query is the set of objects that are associated to some attributes, as can be seen in Google. It is very flexible because attributes can be queried in any order and any combination. However, it is not progressive at all because it dumps all objects that answer the query without organizing them, and it gives no hints on how the query could be refined. So, there is no notion of place, hence no real navigation.

The goal of logic information systems (LIS, for short) is to get the best of both worlds: flexibility in queries, a clear notion of place, and navigation.

2.1 Formal Framework

This section presents the outline of the theory of LIS. Full details can be found in [4].

The contents of a LIS is a set of objects, O, and a mapping d (for *description*) from objects to properties. Objects can be files, like photos and source files, or parts of files, like procedures in a source file. Properties are expressed as formulas of some logic $\mathcal{L}$. Very little is demanded on the logic; its entailment relation (written $\models$ in the sequel) must handle a form of conjunction and disjunction (written $\wedge$ and $\vee$), have a tautology (written $\top$), and be monotonic and decidable. Since the LIS theory was inspired by *formal concept analysis* (FCA [6]), the *object* $\mapsto$ *property* mapping is called a *context*. The logic $\mathcal{L}$ used for expressing properties is a generic parameter of the theory. In practice, well-known logics like proposition logic are not used alone; very specific logics, e.g. for comparing dates, are much more useful. A logic of containment of sets of keywords is often good enough. In this case, LIS comes very close to standard FCA. In standard FCA a context may be infinite, but in LIS it will always be finite.

For convenience, we will say that "an object entails a property p" if its description entails p. The theory of FCA defines the intention of a set of objects as the most specific property that describes *them all*. Dually, the extension of a property is the set of *all* objects that entail the property.

Definition 1 (Intention and extension).

$$\begin{array}{lll} int(O) & = \bigvee_{o \in O} d(o) & \text{where } O \subseteq \mathcal{O} \\ ext(p) & = \{o \in \mathcal{O} \mid d(o) \models p\} & \text{where } p \in \mathcal{L} \end{array}$$

Clearly, intention and extension map set of objects to properties and back.

In fact, $ext \circ int$ and $int \circ ext$ are called *closures*; they are normalizing operators. Note that the normal form of a set of objects or of a property depends on the context. In general, not every subset of O is normal, and there are much less normal subsets than there are subsets. Similarly, not every property is normal.

This shows that pairs $\langle ext(p), int(ext(p))\rangle$ and $\langle ext(int(O)), int(O)\rangle$ are special. They are extension-intention pairs in which the intention is the intention of the extension, and the extension is the extension of the intention. FCA theory calls these pairs *concepts*. Concepts can be ordered: a concept is smaller than another if its extension is contained in the other, or equivalently if its intention entails the other. In LIS, there are always finitely many concepts.

Definition 2 (concept). *A pair $\langle e, i\rangle$ is a* concept *iff $ext(i) = e$ and $int(e) = i$. A concept $\langle e, i\rangle$ is smaller than or equal to a concept $\langle e', i'\rangle$, $\langle e, i\rangle \leq \langle e', i'\rangle$, iff $e \subseteq e'$ (or equivalently $i \models i'$).*

The main result of FCA is that given a context, the set of all its concepts ordered by $\leq$ forms a complete lattice: the *concept lattice*. Furthermore, the original context can be reconstructed from the concept lattice. In some sense, the context is the concrete data-structure, and the concept lattice is the information it contains. In LIS words, the contents of a LIS is a context, but LIS shows concepts as places, and the $\leq$ relation as a subdirectory relation.

$\langle ext(int(\{o\})), int(\{o\})\rangle$ is always a concept, but the converse is not true, simply because there may be more concepts than objects. This concept, written $conceptof(o)$ in the sequel, is said to be *labelled by o*. Similarly, $\langle ext(p), int(ext(p))\rangle$ is always a concept, but it depends on the logic whether there are more concepts than formulas (modulo entailment). It is written $conceptof(p)$, and we say that p labels this concept. The practical interest of this notion is that $\{o\}$ is usually much smaller than $ext(int(o))$, and p is also usually much simpler that $int(ext(p))$.

Finally, one says that properties that label the same concept are *contextually equivalent*. This shows three levels of information in a LIS:

1. Absolute truth is represented as the logic of properties. At this level, $a \wedge b \models a$ or *fish* $\models$ *vertebrate* whatever happens in the context. It is up to the administrator of a LIS to choose what absolute truth he wants to represent.
2. Facts are represented in the context. New facts may be added, old facts may be deleted.
3. Concepts, and contextual entailment, represent contingent truth, i.e. conclusions that may not be valid in another context.

In his logic modeling of information systems querying, Van Rijsbergen [20] proposed that the answer to a query be its extension. However, an extension may be large, especially if the query is vague. So, LIS takes another stand-point [4]:

Definition 3 (conceptual navigation). *The answer to a query q is $dirs(q) \cup files(q)$ where:*

$$\begin{array}{ll} dirs(q) & = \text{a finite set } P \text{ such that} \\ & \quad \forall c < conceptof(q). \left[\begin{array}{l} \neg \exists c' < conceptof(q). c < c' \\ \Rightarrow \exists p \in P. conceptof(q \wedge p) = c \end{array} \right. \\ files(q) & = \{o \mid conceptof(o) = conceptof(q)\} \end{array}$$

The $files(q)$ are the objects at place q, i.e. the files in directory q, whereas the $dirs(q)$ are properties that reach the greatest lower concepts, i.e. the subdirectories of q. Elements of $dirs(q)$ are also called *increments* to avoid the connotation of a file system, and to reflect their use in incremental navigation.

Note that the answer to a query contains other queries (see $dirs(q)$). This is analogous to a dialog between a customer (C) and a shop assistant (SA):

C: I want to buy flowers! What do you have?
SA: Do you have any idea of the color, kind of flower or size of bouquet?
C: I want a big bouquet! What color do you have?
SA: Red, white or yellow.
...

The user never has to guess a formula; he only has to select a formula among the $dirs(q)$ and keep on repeating the process until he finds the object he was looking for. So, a user may even navigate in a context with a logic he does not know, provided he understands the formula. In other words, a passive knowledge of the logic is enough. However, if he has a deeper knowledge of the logic, he can go faster by setting his initial query q directly to a formula that caracterizes the desired object.

To sum it up, a LIS behaves like a schemaless database, its organization structure being computed dynamically from the logical descriptions of the objects. In a database system as in a LIS, queries are intentional. The most striking difference between a database system and a LIS is the nature of the answer. Whereas it is *extensional* in a database system, i.e. objects, in a LIS the answer to a query is also *intentional*, i.e. expressed in the same language as queries. Therefore, the search can be progressive through an iteration process: ask for a query, pick up an answer to complement the query, etc.

2.2 A LIS File System — LISFS

LIS is a candidate for replacing the traditional hierarchical organization when it does not fit well the needs of an application. In fact, a hierarchical organization is often used as a default solution, despite its lack of flexibility. Many applications contain their own browser in order to circumvent the inadequacy of an underlying hierarchical store. Having a LIS as a file system would provide an adequate solution that is ready for use by different applications. As a bonus, this would help making these applications communicate through the file system.

So, we have studied an implementation of LIS as a file system in order to offer a generic service that could be used in already existing applications [13]. This implementation, named LISFS, uses as much as possible database and file system techniques. It is still a prototype but its current state shows acceptable performance for interactive usage with more than 100,000 files.

In modern operating systems, a file system is an implementation of some file system interface (e.g. VFS for Linux, *virtual file system*). Although a file system is best presented at this level, this would reveal technical details that are irrelevant in this article. So, we choose to present LISFS at the shell level. We have not written a new shell; we simply run an existing shell on LISFS.

In LISFS, a fragment of propositional logic with valued attributes is proposed as a kernel logic. To achieve this, paths are considered as formulas; a path is a conjunction of atomic properties (directories)[2], i.e. the UNIX path separator `/` is to be read as a logic conjunction $\wedge$. This gives a new semantics to old shell commands.

EXAMPLE 1 (USING LISFS).

```
[1] mkdir a; mkdir b; mkdir c; mkdir d
[2] touch a/b/d/fabd; touch d/c/b/fbcd; touch d/b/fbd
[3] cd b; ls
    a/ c/ fbd
[4] cd a; ls
    fabd
```

Under LISFS, the `mkdir` *command creates axioms. So at Line 1, four directories are created at the root (*`pwd` $= \top$*), i.e. axioms* `a` $\models \top$*, etc. are declared. Then, three files are created at Line 2 with the standard UNIX command* `touch`*:* `fabd` *has property* `a` $\wedge$ `b` $\wedge$ `d`*,* `fbcd` *has property* `b` $\wedge$ `c` $\wedge$ `d`*, and* `fbd` *has property* `b` $\wedge$ `d`*.*

Starting from the root, command `cd` *is used to go into subdirectory* `b` *at Line 3. Adopting the LIS point of view, this reads: property* `b` *is selected.* `pwd` *becomes* $\top \wedge$ `b` $=$ `b`*.*

This directory contains two subdirectories, `a/` *and* `c/`*, and a file* `fbd`*. Though* `b` $\wedge$ `d` *is not equivalent to* `b`*, it is contextually equivalent. At Line 4, property* `a/` *is selected, setting* `pwd` *to* `a` $\wedge$ `b`*. Only one file remains, namely* `fabd`*.*

Moreover, LISFS can be extended by using a *plugin* mechanism. In this way, LISFS maintains the genericity of the design of LIS. This mechanism also handles to kinds of plugins : *logics* are used to attach new specific logics to some properties (e.g. interval logics or date logics) and *transducers* are programs that can extract properties from file contents as soon as a file is updated. So, the description of a file can be made of both *intrinsic* properties computed and maintained by transducers, and *extrinsic* properties maintained by the user.

Another very important feature of LISFS is that it may operate at two levels. At the first level, called *interfile*, objects are files. At the second level, called *intrafile*, objects are parts of files [14]. The first level is the standard operation level of file systems. The second level permits to treat a file as a directory and explore its constituents. The extension of the `pwd` is always a selection of parts of the original file. In every directory, it is presented as the unique inhabitant of directory `pwd`. At the intra-file level, the extension can be considered as a *slice* or a *view* of the original file for some property. Moreover, these *views* can be edited; the modifications will be retropropagated to the original file, and to every other view that shares something with this one. This unifies access methods inter- and intrafiles, and it makes tools designed to browse a collection of files suitable to browse the components of a file.

2.3 Applications

We have experimented with applications ranging from personal information systems for managing recipes, agenda, music files, photos, one's homedir, to software engineering tools and office applications for managing bibliographies and emails. We are also currently developing a geographic information system prototype.

3. SOFTWARE APPLICATIONS OF LIS

3.1 Indexing Software Components

Component retrieval is a key issue for the hability to reuse components. Prieto-Díaz explains that this implies classifying components, and that it can be done in two ways: hierarchical or faceted classification [15]. He further explains that faceted classifications are more suitable for classifying software components, though they require the intervention of an expert, and they do not work well for heterogenous collections. The requirement for an expert can be avoided if facets are extracted automatically. Furthermore, LIS are designed for coping with heterogenous data. So, we propose to use LISFS as a storage device for a component manager.

Designing such an application amounts to designing a logic for representing component properties, and the structure of the context. The attachment of a property to every single component is automated by a transducer, and LISFS supports the run-time retrieval system. This approach can be transposed to any kind of software components like Web services [12, 9] or COTS [17]. Section 4 presents an application of this approach to Java methods.

3.2 Browsing Source Trees and Source Files

LISFS can be used to browse source trees and sources files.

Regarding source trees, LISFS has been used to manage the Linux kernel source tree. In this case, intrinsic properties of the files are the names of the functions they contain (as extracted with the ctags command). The files also contain extrinsic properties that correspond to their path in the original source tree (e.g. `driver`, `fs`, `include`, etc.).

With such a huge source tree, one would like to classify the files according to his current needs: by file type (includes, C files, text documentation, ...), by architecture (`x86`, `PPC`, ...), by module (drivers, virtual memory manager, ...) and so on. However, the

[2]In fact, a path is a conjunction of formulas that can be atomic, disjunctive (e.g. `small|large`), conjunctive (e.g. `black&blue`), or negative (e.g. `!costly`). The symbol `&` is only necessary for conjunctions that are nested in disjunctions or negations

organization of the Linux kernel source tree suffers from the limitations of hierarchical file systems and therefore forces the developer into using a fixed organization scheme (currently, by type, then by module, then by architecture). Some directories therefore appear in different places (e.g. there are 4 `sound` directories), the organization is not always coherent and is hard to understand. One answer to these problems is to use specific tools, such as LXR [10], a cross-referencing tool for relatively large code repositories.

Another solution is to use LISFS, with which these problems become irrelevant. The classification scheme is not fixed anymore, and the user can select the set of files dealing with memory handling, should they be includes, C files or documentation with `cd mm`. Or he could have retrieved every include files with the same ease: `cd kind:header`, focused on a particular architecture: `cd architecture:ppc`. Of course, those queries can be issued in any order, and in any place ranging from the root of the source tree to the deepest subdirectory where it is still relevant.

Once a source file is located in the tree, the programmer is able to work on it. However, a source code contains scattered information, such as debugging statements, comments, assertions, etc. As recognized by Aspect-Oriented Programming [8], this cross-cutting information is generally not handled in an easy way because it is scattered in the program. With LISFS intra-file mode, it becomes easy to manage such cross-cutting concerns.

To do so, the user has to input the special query `cd parts`; then, the file is diplayed along with several subdirectories, each of them leading to a partial view of the initial file. For this to work under LISFS, plugins need to be designed, one for each kind of programming languages. Such plugins have been developed for several languages, among which C source files (used when working on the Linux kernel sources) and OCaml source files (used when working on LISFS source code).

Therefore, we are able to ask for a partial view of source files written in those languages. E.g., we can ask for a view of a source file with comments and without debugging statements by issuing the query `cd comment|(!aspect:debug)`. The result is a place that holds a copy of the file where the non-selected parts (here, the debug statements) are hidden. Any change operated on this partial view will be propagated in the original file. Moreover, the result of an `ls` command shows other subdirectories, i.e. it prints out how to get smaller views of the source file. In the case of C files, it shows we can focus on a particular `function:`, or make a slice regarding a particular variable (`var:`), and so on.

3.3 Analysis of Program Traces

LISFS can be used to explore execution traces of programs.

In a first approach, we focus on Prolog program execution traces. Every event of a trace is described by a text line that mentions the values of a set of attributes (event number, port, predicate, goal, depth) reflecting Byrd's box model [1]. Using LISFS intrafile level, a trace can form a context where events are objects to be browsed. For instance, `cd depth:>5` gives a partial view of the trace showing only goals whose depth is greater than 5.

Another approach is to query a pool of different execution traces of the same program corresponding to different test cases, or of different execution traces of different mutants on the same test case. In this case, traces are the objects of the context. They are described by test verdicts (*pass* or *fail*), and the numbers of the executed lines, or other trace information. The problem of locating bugs using traces and test verdicts is not new [7], but it was recently rephrased in the context of data-mining [2]. The advantage is that data-mining them gives well-known indicators (e.g. support, confidence, lift) that can be used in a more principled way than the *ad hoc* indicators used by Jones *et al.*

We propose to use LISFS to crosscheck the traces so as to locate errors in the code. A further advantage of using LIS instead of the standard association rules method is that LIS can accommodate a large range of logical descriptions whereas association rules are limited to attribute-based contexts. So, we expect to be able to use other trace descriptors than merely line numbers. Preliminary experiments are currently being lead on C programs execution traces.

4. SOFTWARE COMPONENT INDEXING

We present more details on the application for indexing software components. In this section, components are Java methods.

4.1 A Logic for Classifying Java Methods

We classify facets as *formal*, *semi-formal*, and *informal*. We did not try to implement every possible facet, but we decided to implement one in each kind of facets. This shows how very different kinds of descriptions combine in a single property, and are used in navigation.

Formal facets are formally related to the semantics of components; we have chosen types because they are already given in the source, though any static property would do. Concerning type, we have developed an entailment relation which combines type isomorphisms [3] with the inheritance relation. This is a contribution in itself since prior attempts to do so in the higher-order type case lead to a contradiction [18], while we show that the first-order type case works well. From a LIS perspective this also shows how very specific logics can be developed to fit a given purpose.

Semi-formal facets are only loosely connected to the semantics; we have chosen methods identifiers, and especially the convention that helps splitting an identifier into a phrase. For instance, `getValue` is usually meant to be read as "get value". This reading is not connected to the semantics because nothing forces a method `getValue` to get anything, and *vice versa*. However, identifiers are formally connected to the semantics by the store; a spelling mistake is seen by the compiler. In a given context, whatever `getValue` means is formally defined.

On the opposite, we also developed an informal facet based on comments; they are not formally connected to the semantics at all, and a spelling mistake is not seen by the compiler.

We propose a type entailment relation that is based on inheritance, written $t \leq_{\text{inh}} t'$ iff t inherits from t', and on type isomorphism, written $t \sim_T t'$ iff t is isomorphic to t' wrt. theory T. We consider Java as a first-order object-oriented language in which the only polymorphism comes from inheritance; there is no polymorphism *à la* ML. The language of Java types is abstracted as follows:

DEFINITION 4 (TYPES).

$$\begin{array}{lcl} \mathit{Type} & ::= & \mathit{Arg} \to \mathit{Res} \\ \mathit{Arg} & ::= & \mathit{Arg} \times \mathit{Arg} \mid \mathit{class} \\ \mathit{Res} & ::= & \mathit{class} \end{array}$$

We write $x : t$ iff x has type t.

In order to use types as descriptions of objects in LIS, hence as queries, we have to define what is the entailment relation of types considered as a logic. First, we show intuitively that the arrow type must be *contravariant* for the entailment relation.

DEFINITION 5 (CONTRAVARIANCE). *Let $\leq$ be a partial order defined on types as in Definition 4. Relation $\leq$ is said to be contravariant iff*

$$\forall \sigma, \sigma', \tau, \tau', \sigma' \leq \sigma, \tau \leq \tau' \quad \textit{implies} \quad \sigma \to \tau \leq \sigma' \to \tau'$$

Indeed, any entailment relation can be considered as a partial order. Moreover, type entailment must extend the inheritance relation, i.e. $t \leq_{\text{inh}} t' \Longrightarrow t \models t'$, because it is interpreted as $t \leq_{\text{inh}} t' \Longrightarrow ext(t) \subseteq ext(t')$. For instance, `Number` $\leq_{\text{inh}}$ `Object` means that every `Number` is an `Object`, but the opposite is false. Finally, the arrow type must be considered as contravariant for the inheritance relation, hence it must be considered as contravariant for the entailment relation too because the latter extends the former. Why must the arrow type be considered as contravariant for the inheritance relation? Let us take an example. Assume a user is looking for a method that takes a `Button` as a parameter and returns a `Container`. His query will be `Button` $\rightarrow$ `Container`. Every method that accepts a super-type of `Button`, plus supplementary parameters, and that returns a subtype of `Container` is a correct answer. This shows that parameters and results play opposite roles. In fact, type entailment means "can replace the other, and still be type-checked".

When taken literally, types are not good search keys because two types may differ though semantically neutral transformations would make them identical. For instance, the order of parameters should not matter because a parameter permutation makes types identical. This is called *type isomorphism*, and has been recognized for long as the key to use types as queries for searching software components [16, 18].

Definition 6 (Type isomorphism). *A type t is isomorphic to a type t', written $t \sim t'$, iff*

$$\exists f : t \rightarrow t' \quad \exists g : t' \rightarrow t \quad [g \circ f = \mathrm{Id}_t \wedge f \circ g = \mathrm{Id}_{t'}]$$

Functions f and g are called the witnesses *of the isomorphism. They are the semantically neutral operations that make two types equivalent.*

It is convenient to present type isomorphisms as equivalence relations wrt. a set of axioms. Every set of axioms generates an isomorphism. For instance, axiom $\sim_{\text{exch}}$ (exchange) says that $t \times t'$ and $t' \times t$ must be considered as isomorphic, and axiom $\sim_{\text{curry}}$ (curryfication) says that $t \rightarrow (u \rightarrow v)$ and $(t \times u) \rightarrow v$ are isomorphic. Isomorphism axioms generate equivalence classes that make a type the representent of each type of its class. Di Cosmo has developed a complete theory of isomorphism axioms [3], however $\sim_{\text{exch}}$ is the only axiom that is relevant to our type language. Other axioms, like $\sim_{\text{curry}}$, deal with characteristics of other type systems like higher-order.

The entailment relation can also be presented as a set of axioms. We already know the $\models_{\text{inh}}$ axiom, which says that entailment extends inheritance. A second axiom is $\models_{\text{drop}}$, which says that $(t \times u) \rightarrow v \models t \rightarrow v$. Other axioms have been developed in the literature, but they do not apply to our type language; e.g., $\models_{\text{inst}}$ which says $\forall \alpha.t \models t[\alpha \leftarrow t']$ applies to polymorphism *à la* ML.

Finally, an entailment relation can be built by combining isomorphism axioms, hence considering equivalent classes of types, and entailment axioms. However, entailment axioms may generate new equivalence classes, e.g. if $t \models t'$ and $t' \models t$. It may even make the entire set of types collapse into too few equivalent classes. This is what happens when isomorphism axiom $\sim_{\text{curry}}$ and entailment axioms $\models_{\text{drop}}$ and $\models_{\text{inst}}$ are combined [18]. So, it is important to study what happens with axioms that apply to our type language: $\sim_{\text{exch}}$, $\models_{\text{drop}}$ and $\models_{\text{inh}}$. In fact, we have proven that every equivalence class induced by $\sim_{\text{exch}}$, $\models_{\text{drop}}$ and $\models_{\text{inh}}$ is already induced by $\sim_{\text{exch}}$.

Theorem 1 (Soundness of $\sim_{\text{exch}}$,$\models_{\text{drop}}$ and $\models_{\text{inh}}$).

$$\forall t, t' \left[\begin{array}{l} \left[\begin{array}{c} t \models t' \text{ wrt. } \sim_{\text{exch}},\ \models_{\text{drop}}, \text{ and } \models_{\text{inh}} \\ \wedge \\ t' \models t \text{ wrt. } \sim_{\text{exch}},\ \models_{\text{drop}}, \text{ and } \models_{\text{inh}} \end{array} \right] \\ \Longrightarrow t \sim t' \text{ wrt. } \sim_{\text{exch}} \end{array} \right]$$

To conclude, we choose as a logic for representing types the entailment relation induced by axioms $\sim_{\text{exch}}$, $\models_{\text{drop}}$ and $\models_{\text{inh}}$.

The entailment relation for keywords is much simpler. Semi-formal and informal properties are sets of keywords, and we say that $s \models s'$ iff $s \supset s'$.

4.2 Implementation

The previous section has shown the logical engineering one must engage into to develop a formal method. It is not related to LISFS. This section will show how LISFS helps in implementing the resulting entailment relation at almost no cost.

We could have developed a logic plugin for type entailment, but LISFS allows an alternative solution. The native solver of LISFS already implements a fragment of propositional logic that contains the exchange rule, $\frac{A \wedge B}{B \wedge A}$, and the weakening rule, $\frac{A \wedge B}{A}$. These rules correspond to axioms $\sim_{\text{exch}}$ and $\models_{\text{drop}}$ at the propositional level. So, we propose to represent types in such a way that exchange and weakening will implement axioms $\sim_{\text{exch}}$ and $\models_{\text{drop}}$.

Contravariance tells that types behave differently according to their context. We call *positive* the types of the right end of $\rightarrow$, and *negative* the types of the left end. Similarly, exception types are called positive, because they behave as results, and class types are called negative, because they behave as parameters. So, we represent complex types as conjunctions of signed base types.

Definition 7 (Encoding of types as conjunctions). *We note $\lceil . \rceil$ the encoding of types into conjunctions of signed base types. Remember that $--t = +t = t$.*

$$\begin{array}{lcl} \lceil t \rightarrow t' \rceil & = & -\lceil t \rceil \wedge \lceil t' \rceil \\ \lceil t \times t' \rceil & = & \lceil t \rceil \wedge \lceil t' \rceil \\ \lceil \textit{base type} \rceil & = & \textit{base type} \end{array}$$

Using encoding $\lceil . \rceil$, the implementation of axioms $\sim_{\text{exch}}$ and $\models_{\text{drop}}$ comes for free. What remains to implement is the $\models_{\text{inh}}$ axiom. It is simply done by using LISFS command `mkdir` which implements user-defined axioms (see Section 2.2). Every time a declaration says that a class `A` inherits from a class `B`, symbols `in-A` and `out-A` are created for representing $-$`A` and +`A`, and a command `mkdir out-B/out-A` creates the axiom +`A` $\models$ +`B`. Once all the inheritors of a class `B` are known, say $\mathtt{A}_1, \ldots, \mathtt{A}_n$, a command `mkdir in-`$\mathtt{A}_1$`/.../in-`$\mathtt{A}_n$`/in-B` creates the axioms $-$`B` $\models -\mathtt{A}_1$, $\ldots$, $-$`B` $\models -\mathtt{A}_n$.

All this implements the desired entailment relation without having to develop a theorem prover.

4.3 Experiments

We have experimented our Java method browser on existing Java packages. Consider, for instance, the AWT package. It consists of about 5,200 methods. The context builder passes through the package and creates its lines and columns in two passes: type first, then identifiers and comments. Figure 1 shows the shape of the actual AWT context. Indeed, every symbol has an internal identifier in LISFS (its inode), and the figure simply shows the context as a matrix. The figure can be interpreted as follows.

Objects, i.e. AWT methods, are on the horizontal axis, and attributes on the vertical one. Object numbers are introduced in the order of method declarations in the program. Types are analyzed before identifiers and comments, so they are given the low attribute numbers (below $\approx$1,700 on the figure). Types are numbered during a traversal of the inheritance graph. Once this is done, identifiers and comments are read in the source order, so they are given the highest attribute numbers. The figure therefore displays a type area, and a keyword area. The type area is strongly structured by lines

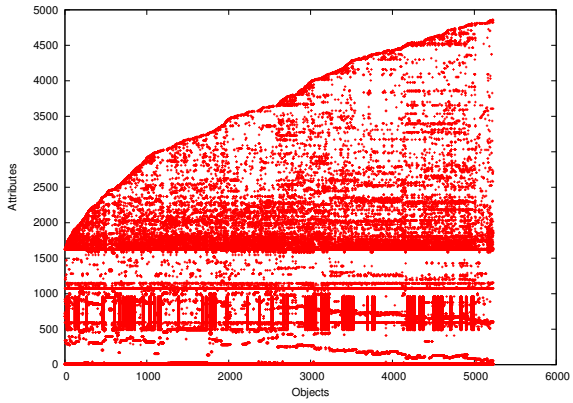


Figure 1: AWT formal context

and columns. Lines show types that are shared by many objects, whereas columns show types that come together in a single object. Columns are mainly an effect of inheritance. The keyword area is the triangular part. It shows that keywords that appear first are also frequent keywords since the triangle basis is darker than its top. The keyword area shows no particular structure. LIS and LISFS propose a rational navigation principle in this kind of structure.

Let us assume that a user looks for a method that takes a string as a parameter. The following simple commands help analyzing the situation.

```
[1] cd /mnt/lisfs/component-manager/
[2] cd in-java.lang.String
[3] ls | wc -l
        444
[4] ls .ext | wc -l
        356
```

We present the queries as shell commands, but one should rather imagine them as buttons of a graphical interface. The last two commands show that there are 356 methods that takes a string as a parameter, and that there at 444 ways to make the user demand more precise. So, the user had better think a little on his own, and he remembers that what he wants to do is related to MIME types.

```
[5] cd 'ident:mime|comment:mime'
[6] ls
     1 in-java.awt.datatransfer.MimeTypeParseException/
     2 in-java.awt.datatransfer.MimeTypeParameterList/
     3 static/
     3 in-java.awt.datatransfer.SystemFlavorMap/
     3 in-java.awt.datatransfer.DataFlavor/
     4 in-java.awt.datatransfer.MimeType/
     5 by-exception/
     7 out/
    11 comment/
    12 ident/
    12 method/
    12 by_class/
    12 access_control/
```

Of the 5,200 methods of the initial context, only 12 are possible answers, and there are few relevant increments. So, the user can study them and recall that the command he is looking for is not static.

```
[7] cd '!static'
[8] cd .ext
[9] ls
    MimeTypeParameterList       MimeTypeParseException
    isMimeTypeEqual             normalizeMimeType
    normalizeMimeTypeParameter  parse
[10] cat normalizeMimeType
    Called for each MIME type string to [...]
```

The user has recognized the name `normalizeMimeType` and checked its summary. Overall, the navigation took 3 steps that invoked the three kinds of properties. Each kind of property determines a classification of Java methods in itself, but it is LIS that combines them all in a single classification.

Former propositions by Rittri, Runciman, and Di Cosmo described classifications based on type isomorphism, but they lacked proper ways to navigate in possible answers. Furthermore, in their propositions the user is to submit a nearly complete expected type. In ours, he only submits the part that seems relevant.

Another possible classification of Java methods is by using the inheritance graph of the classes they belong to. This is the only classification used in the official Java documentation. Assume the developer of a graphical interface looks for a method that would return the name of a window. He may start by exploring the class `Window`.

```
[11] cd /mnt/lisfs/component-manager/class-java.awt.Window
[12] ls
      3  static/
     11 by_exception/
     24 final/
    255 class-java.awt.TextField/
    255 class-java.awt.TextArea/
    255 class-java.awt.Scrollbar/
    255 class-java.awt.List/
    255 class-java.awt.Label/
    255 class-java.awt.Choice/
    255 class-java.awt.Checkbox/
    255 class-java.awt.Canvas/
    255 class-java.awt.Button/
    328 comment/
    361 class-java.awt.ScrollPane/
    361 class-java.awt.Panel/
    361 class-java.awt.Container/
    438 ident/
    438 method/
    438 by_type/
    438 access_control/
[13] ls .ext | wc -l
       438
```

LISFS computes 20 increments in a few seconds. They concern 438 AWT methods. Since he is looking for window names, the developer searches for keyword “name”.

```
[14] cd ident:name
[15] ls
    1 ident:set/
    1 ident:get/
    2 ident:construct/
    2 ident:component/
    3 class-java.awt.TextField/
    3 class-java.awt.TextArea/
    3 class-java.awt.Scrollbar/
    3 class-java.awt.ScrollPane/
    3 class-java.awt.Panel/
    3 class-java.awt.List/
    3 class-java.awt.Label/
    3 class-java.awt.Container/
    3 class-java.awt.Choice/
    3 class-java.awt.Checkbox/
    3 class-java.awt.Canvas/
    3 class-java.awt.Button/
    4 comment/
    4 by_type/
    4 access_control/
```

Keyword “get” seems relevant. A quick look will confirm it.

```
[16] cd ident:get
[17] ls
     getName
[18] cat .ext/getName
    Gets the name of the component.
```

These experiments confirm the feasibility and interest of combining a wide range of properties. The user submits queries using properties that are relevant for him, and LIS return relevant increments. The navigation may start by submitting type information and continue using keywords; the user is not bound to using a single classification.

5. CONCLUSION

LIS and LISFS form a flexible and powerful framework for developing software engineering tools. We have already made experiments in source browsing, component browsing, and trace analysis for bug finding. As it includes basic logic services, LISFS can also help in the logical engineering of an application.

We have also presented the design of a component browsing tool that we have demonstrated on real Java packages.

In its current state, LISFS can only handle contexts in which properties are attached to one object at a time; in other words, properties are unary predicates. We are developing a variant of LIS in which properties can be attached to vectors of objects [5]. So doing, properties are n-ary predicates. This is especially important in software engineering applications because they are rich in inter-objects relations, e.g. calls, imports, compiles-to, inherits-from, etc. These relations can be simulated with unary predicates, but it is inconvenient. For instance, maintaining an inverse relation is error-prone.

The concept lattice is based on strict containment, so that navigation increments are always relevant. However, it could be useful to consider qualified containment like 90% of concept c belongs to concept c'. This leads to introducing *association rules* and data-mining operations in LIS. We believe it is useful in exploratory applications like fault-finding [2].

LIS and LISFS propose a navigation metaphor based on the notion of place through the use of increments; in particular the concept lattice is never actually built. Other authors have proposed to navigate graphically in the concept lattice. We believe this is impracticable because there are too many concepts, even in not so large contexts. For instance, the AWT context produces about 135,000 concepts. However, what LISFS computes is only a path through it.

Facetted classification *à la* Prieto-Díaz and type isomorphisms *à la* Di Cosmo are opposite methods for component finding. The former is informal and manual, while the latter is formal and automated. However, LISFS combines both approaches. We believe it is a step towards an answer to Mili *et al.*'s remark, "*no solution offers the right combination of efficiency, accuracy, user-friendliness and generality to afford us a breakthrough in the practice of software reuse*." [11]

generality: On the one hand, LIS as a framework is generic with respect to the logic used in object descriptions. There is no prerequisite on classifications, and on their number. On the other hand, LISFS as a file system integrates well with other tools. For instance, file browsers automatically become concept browsers when mounted on a LISFS partition.

user-friendliness: Because it is generic, LIS can adapt to the user's preferred logic. Furthermore, it proposes a dialog-based navigation in which the user needs only a passive knowledge of the description language. This makes it easy to grasp LIS progressively. As a file system, LISFS offers nothing directly but can be easily used through a graphical interface.

accuracy: As a formal framework, LIS behaviour is completely defined. In particular, navigation is complete in a formal sense (i.e. every object can be accessed using only increments returned by LIS; the user need not invent queries), and increments are always relevant (i.e. they never lead to dead-ends, and they are always focusing on a strict subconcept of the current concept). Combining classifications also increases accuracy because an application based on several classifications will always be at least as accurate as the most accurate of the classifications. So, if a classification is good for homogenous contexts and another is good for heterogenous contexts, the combination of both will be good in both cases.

efficiency: LISFS is an efficient implementation of LIS that can handle up to 100,000 objects with reasonable performance. This is still less efficient than an ordinary file system or database. However, it offers more services and it is still in its infancy; hierarchical file systems and databases have been improved by about 30 years of intensive usage. We believe LISFS can improve its peformance to the level of state-of-the-art file systems.

6. REFERENCES

[1] L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-Å. Tärnlund, editor, *Proc. of the Logic Programming Workshop*, Debrecen, 1980.

[2] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces. A re-interpretation of Jones, Harrold and Stasko test information visualization. In T. Ellman and A. Zisman, editors, *20th Int. Conf. on Automated Software Engineering*. ACM Press, 2005.

[3] R. Di Cosmo. Deciding type isomorphisms in a type-assignment framework. *Journal of Functional Programming*, 3(4):485–525, 1993.

[4] S. Ferré and O. Ridoux. Introduction to logical information systems. *Information Processing and Management*, 40(3):383–419, 2004.

[5] S. Ferré, O. Ridoux, and B. Sigonneau. Arbitrary relations in formal concept analysis and logical information systems. In F. Dau, M.-L. Mugnier, and G. Stumme, editors, *ICCS*, volume 3596 of *LNCS*. Springer, 2005.

[6] B. Ganter and R. Wille. *Formal concept analysis — Mathematical Foundations*. Springer, 1999.

[7] J. Jones, M. J. Harrold and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Int. Conf. on Software Engineering*, 2002.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, volume 1241 of *LNCS*. Springer-Verlag, 1997.

[9] J. Liberty. *Programming C#*. O'Reilly, 2001.

[10] Linux cross-reference project. Available on `http://lxr.linux.no/`.

[11] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349–414, 1998.

[12] S. Overhage and P. Thomas. WS-Specification: Specifying web services using UDDI improvements. In *Web, Web-Services, and Database Systems*, volume 2593 of *LNCS*. Springer, 2003.

[13] Y. Padioleau and O. Ridoux. A logic file system. In *USENIX Annual Technical Conference*, 2003.

[14] Y. Padioleau and O. Ridoux. A parts-of-file file system. In *USENIX Annual Technical Conference*, 2005.

[15] R. Prieto-Díaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[16] M. Rittri. Using types as search keys in function libraries. In *4th Int. Conf. on Functional Programming Languages and Computer Architecture*. ACM Press, 1989.

[17] G. Ruhe. Intelligent support for selection of COTS products. In *Web, Web-Services, and Database Systems*, volume 2593 of *LNCS*. Springer, 2003.

[18] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. In *4th Int. Conf. on Functional Programming Languages and Computer Architecture*. ACM Press, 1989.

[19] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[20] C. J. van Rijsbergen. A new theoretical framework for information retrieval. In *Int. Conf. on Research and Development in Information Retrieval*, 1986.

Mining Eclipse for Cross-Cutting Concerns

Silvia Breu
University of Cambridge
Computer Laboratory
Cambridge, UK
silvia@ieee.org

Thomas Zimmermann
Saarland University
Dept. of Computer Science
Saarbrücken, Germany
tz@acm.org

Christian Lindig
Saarland University
Dept. of Computer Science
Saarbrücken, Germany
lindig@cs.uni-sb.de

ABSTRACT

Software may contain functionality that does not align with its architecture. Such cross-cutting concerns do not exist from the beginning but emerge over time. By analysing where developers add code to a program, our history-based mining identifies cross-cutting concerns in a two-step process. First, we mine CVS archives for sets of methods where a call to a specific single method was added. In a second step, such simple cross-cutting concerns are combined to complex cross-cutting concerns. To compute these efficiently, we apply formal concept analysis—an algebraic theory. History-based mining scales well: we are the first to report aspects mined from an industrial-sized project like Eclipse. For example, we identified a locking concern that crosscuts 1284 methods.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

General Terms

Languages, Documentation, Algorithms

1. INTRODUCTION

As object-oriented programs evolve over time, they may suffer from *"the tyranny of dominant decomposition"* [15]: The program can be modularised in only one way at a time. Concerns that are added later and that no longer align with that modularisation end up scattered across many modules and tangled with one another. Aspect-oriented programming (AOP) remedies this by factoring out aspects and weaving them back in a separate processing step [7]. For existing projects to benefit from AOP, these cross-cutting concerns must be identified first. This task is called *aspect mining*.

MSR'06, May 22–23, 2006, Shanghai, China.

We solve this problem by taking a historical perspective: Our analysis is based on the hypothesis that cross-cutting concerns are added to a project over time. A code change in the history of a program is likely to introduce such a concern if the modification gets introduced to various locations within a single code change. This observation is our conceptual contribution.

Our hypothesis is supported by the following example: On November 10, 2004, Silenio Quarti committed code changes "76595 (new lock)" to the Eclipse CVS repository. These changes fixed the bug #76595 "Hang in gfk_pixbuf_new" that reported a deadlock[1] and required the implementation of a new locking mechanism for several platforms. The extent of Silenio Quarti's modification was immense: He modified 2 573 methods and inserted in 1 284 methods a call to the `lock` method, as well as a call to an `unlock` method. Obviously AOP could have been used to weave in this locking mechanism.

For the locking mechanism of Eclipse, it turns out that the locations where calls to `lock` were inserted are exactly the same as the locations where calls to `unlock` were added. This is why we combine the two simple aspect candidates into a *complex aspect candidate*: `lock`, `unlock` were added in 1 284 different locations. However, in the presence of many complex aspect candidates it is not obvious how to find them efficiently. We propose to use *formal concept analysis* [4] for automatically detecting complex aspect candidates, which is our technical contribution and detailed in the next section.

2. MINING CROSS-CUTTING CONCERNS

Previous approaches to aspect mining considered only a single version of a program using static and dynamic program analysis techniques. We introduce an additional dimension: the *history* of a project. Technically, we mine version archives for aspect candidates.

We model the history of a program as a sequence of transactions. A *transaction* collects all code changes between two versions, called *snapshots*, made by a programmer to complete a single development task. Within each transaction we are searching for *added method calls* which may identify an aspect. We consider calls to a small set of (related) methods that are added in many (unrelated) locations a cross-cutting concern or *aspect candidate*.

We refer to a method where calls are added as *location*, and to the method being called simply as *method*. An aspect candidate is thus characterised by two sets: a set of

[1] `https://bugs.eclipse.org/`

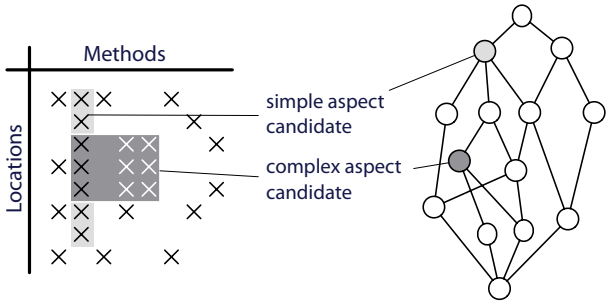


Figure 1: Maximal blocks represent aspect candidates in a transaction (left). Here, 14 candidates form a lattice of super and sub aspects (right). A sub aspect (dark) crosscuts fewer locations but calls more methods than a super aspect (light).

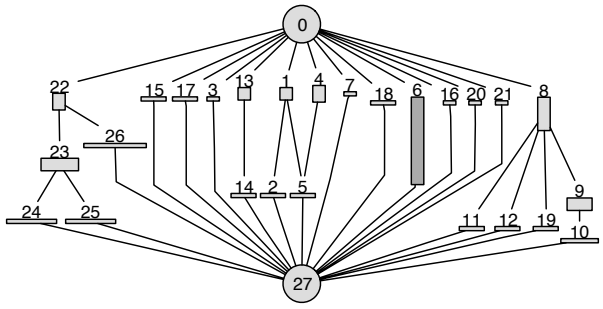


Figure 2: The lattice of aspect candidates from a commit to Eclipse CVS on 2004-03-01 by developer `ptff`. Candidate 6 contains 14 additions of calls to `unsupportedIn2()`.

locations and a set of methods. This definition represents a trade-off: albeit it is not fully general, it still captures many interesting cross-cutting concerns and enables us to identify them efficiently.

Aspects are maximal Blocks. We can think of a transaction as a cross table with locations as rows and methods as columns (Figure 1, left). The intersection of location l and method m is marked with a cross when the transaction inserts a call to m in location l. In this representation, each column is a simple aspect candidate; however, to cut out noise, we only consider columns with at least 7 crosses. Formally, a candidate is a pair (L, M) of locations L and methods M with $|M| = 1$ and $|L| \geq 7$ for simple candidates.

Given a specific simple aspect candidate (L, M), we can arrange the table such that all rows from L are adjacent to each other. Now a simple aspect candidate manifests itself as a *maximal block* in the table of width $|M| = 1$ and height $|L|$. In Figure 1 such a block is marked by the grey-shaded rectangle of size 1×7. A *complex aspect candidate* (L, M) is a maximal block with $|M| > 1$: At each location $l \in L$ all methods $m \in M$ are called. An example is the second dark-grey-shaded rectangle of size 3×3 in Figure 1. However, to obtain such a block for a complex aspect candidate in general, we have to re-order not just rows but also columns. It is therefore not obvious how to compute all blocks present in a transaction.

Identifying maximal blocks in a cross table (or transaction) $T \subseteq \mathcal{L} \times \mathcal{M}$ is provided by the algebraic theory of formal concepts [4]. A maximal block is a pair (L, M) where the following holds:

$$\begin{aligned} L &= \{l \in \mathcal{L} \mid (m, l) \text{ for all } m \in M\} \\ M &= \{m \in \mathcal{M} \mid (m, l) \text{ for all } l \in L\} \end{aligned}$$

Each block (L, M) is maximal in the following sense: we can't add another method m to M without shrinking L to ensure that *all* locations in L call m. Likewise, we can't add another location l to L without shrinking M. The definition allows for blocks of any size. However, we only consider blocks with $|L| \geq 7$ as aspect candidates. To identify the most interesting ones, we additionally take the *area* $|L| \times |M|$ of a block as a measure.

In the worst case, a transaction may contain exponentially many blocks. This makes concept analysis potentially expensive–even in the presence of efficient algorithms [9]. This is not a concern here since we compute the blocks for each transaction individually. Computing all blocks for the 43 270 transactions of Eclipse took about 43 seconds, that is, about one millisecond per transaction.

The aspect candidates of a transaction form a lattice given the following partial order: $(L, M) \leq (L', M')$ iff $L \subseteq L'$. A sub aspect cross-cuts fewer locations than its super aspect but calls more methods (c.f. Figure 1, right). In our experience, aspects in one transaction are rarely in a super/sub order but typically unordered.

3. EXAMPLES

Figure 2 shows the lattice of all aspect candidates from an Eclipse CVS commit transaction on 2004-03-01. In the lattice two aspects are connected if they are in a direct super/sub-concept relation. Nodes are given the shape of the corresponding block which gives prominence to large aspect candidates: For example, candidate 6 contains 14 location where calls to `unsupportedIn2()` were added. This method throws an exception if the operation called is not supported at API level 2.0.

```
public void setName(SimpleName name) {
        if (name == null) {
                throw new IllegalArgumentException();
        }
        ASTNode oldChild = this.methodName;
        preReplaceChild(oldChild, name, NAME_PROPERTY);
        this.methodName = name;
        postReplaceChild(oldChild, name, NAME_PROPERTY);
}
```

An even larger example for a cross-cutting concerns is the following: Eclipse represents nodes of abstract syntax trees by the abstract class `ASTNode` and several subclasses. These subclasses fall into the following simplified *categories*: expressions (subclass `Expression`), statements (subclass `Statement`), and types (subclass `Type`). Additionally, each subclass of `ASTNode` has *properties* that cross-cut the class hierarchy. An example for a property is the *name* of a node: There are named (`QualifiedType`) and unnamed types (`PrimitiveType`), as well as named expressions (`FieldAccess`). Additional properties include the *type*,

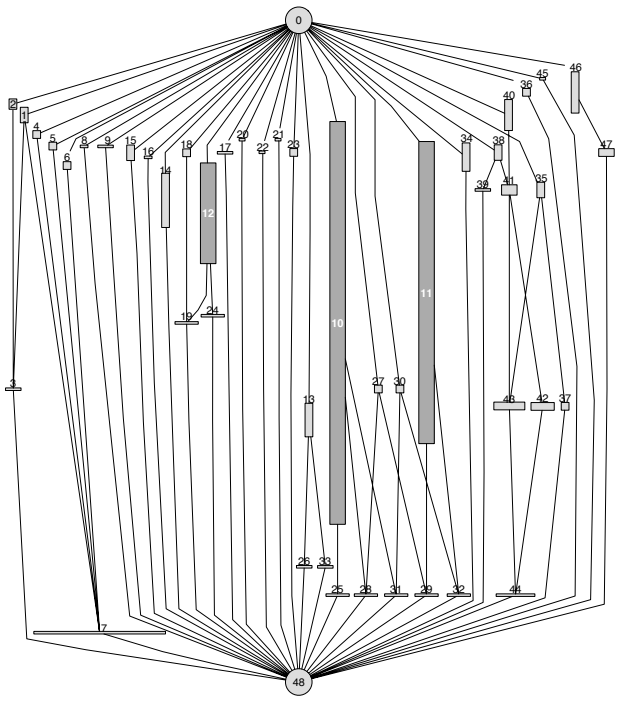


Figure 3: The lattice of aspect candidates from a commit to Eclipse CVS on 2004-02-25 by developer `ptff`. Candidate 10, e.g., contains 104 additions of calls to `preReplaceChild(3)`, `postReplaceChild(3)`.

expression, *operator*, or *body* that are associated with a node in an abstract syntax tree.
This is a typical example for a *role super-imposition* concern [12]. As a result of this cross-cut, every named subclass of `ASTNode` implements the method `setName` which results in duplicated code that is difficult to maintain. With aspect-oriented programming the concern could be realised with the method introduction mechanism.

Our mining approach revealed this cross-cutting concern with several aspect candidates. The lattice for the corresponding commit transaction is shown in Figure 3.

The methods `preReplaceChild` and `postReplaceChild` are called in the aforementioned `setName` method and many other methods. Node 10 contains 104 locations where calls to both methods are added. The methods `preLazyInit` and `postLazyInit` guarantee the safe initialisation of properties and calls to them are added in 78 locations; node 11 is the corresponding node in the lattice in Figure 3. The methods `preValueChange` and `postValueChange` are called when a new operator is set for a node; calls to them have been added in 26 locations, represented by node 12 in the lattice.

4. DATA COLLECTION

Our mining approach can be applied to any version control system, however, we based our implementation on CVS since most open-source projects are using it. One of the major drawbacks of CVS is that commits are split into individual check-ins and have to be reconstructed. For this we use a *sliding time window* approach [20] with a 200 seconds window. A reconstructed commit consists of a set of revisions R where each revision $r \in R$ is the results of a single check-in.

Additionally, we need to compute method calls that have been inserted within a commit operation R. For this, we build abstract syntax trees (ASTs) for every revision $r \in R$ and its predecessor and compute the set of all calls C_1 in r and C_0 for the preprocessor by traversing the ASTs. Then $C_r = C_1 \setminus C_0$ is the set of inserted calls within r; the union of all C_r for $r \in R$ forms a *transaction* $T = \bigcup_{r \in R} C_r$ which serves as input for our aspect mining.

Unlike Williams and Hollingsworth [18, 19], our approach does not build (compile, link) *snapshots* of a system to compute inserted method calls. As they point out, such interactions with the build environment (compilers, make files) are extremely difficult to handle and result in high computational costs. Instead, we analyse only the differences between single revisions. As a result, our preprocessing is cheap, as well as platform- and compiler-independent; the drawback is that types cannot be resolved because only one file is investigated. In particular, we miss the signature of called methods. In order to reduce name collision, we use the number of arguments in addition to method names to identify methods calls. We believe this is good enough because we are analysing one transaction at a time.

5. RELATED WORK

While this work is not the first that applies formal concept analysis as static analysis to mine cross-cutting functionality, it is the first that leverages software repositories to do so. Furthermore, our approach is the first that scales to industrial-sized projects such as Eclipse.

Static Aspect Mining. The Aspect Browser [5] identifies cross-cutting concerns with textual-pattern matching (much like "grep") and highlights them. The Aspect Mining Tool (AMT) [6] combines text- and type-based analysis of source code to reduce false positives. Ophir [14] uses a control-based comparison, applying code clone detection on program dependence graphs. Tourwé and Mens [17] introduce an identifier analysis, that is based on formal concept analysis for mining aspectual views such as structurally related classes and methods. Krinke and Breu [8] propose an automatic static aspect mining based on control flow. The control flow graph of a program is mined for recurring execution patterns of methods. The fan-in analysis by Marin, van Deursen, and Moonen [13] determines methods that are called from many different places—thus having a high fan-in. Our approach presented here is similar to the fan-in analysis. However, with access to serveral versions of a program we can rule out certain such functions as non cross-cutting and therefore are more precise.

Dynamic Aspect Mining. DynAMiT (Dynamic Aspect Mining Tool) [1, 3] is a dynamic approach that analyses program traces reflecting the run-time behaviour of a system in search for recurring execution patterns of method relations. Tonella and Ceccato [16] suggest a technique that applies concept analysis to the relationship between execution traces and executed computational units (methods).

Hybrid Techniques. Loughran and Rashid [11] investigated possible representations of aspects found in a legacy system in order to provide best tool support for aspect mining. Breu also reports on a hybrid approach [2] where the

dynamic information of the previous DynAMiT approach is complemented with static type information such as static object types.

Mining Co-change. One of the most frequently used techniques for mining version archives is co-change. The basic idea is simple: *Two items that are changed together in the same transaction, are related to each other.* Our approach is also based on co-change. However, we use a different, more specific notion of co-change. Methods are part of a (simple) aspect candidate when they are changed together in the same transaction and *additionally the changes are the same*, i.e., a call to the same method is inserted.

Mining Co-addition of Method Calls. Recently, research extended the idea of co-change to *additions* and applied this concept to method calls: *Two method calls that are inserted together in the same transaction, are related to each other.* Williams and Hollingsworth used this observation to mine pairs of functions that form usage patterns from version archives [19]. Livshits and Zimmermann used data mining to locate patterns of arbitrary size and applied dynamic analysis to validate their patterns and identify violations [10]. Our work also investigates the addition of method calls. However, within a transaction, we do not focus on calls that are inserted together, but on locations where the same call is inserted. This allows us to identify cross-cutting concerns rather than usage patterns.

6. CONCLUSIONS

We are the first who leverage version history to mine aspect candidates. Previous approaches considered a program only at a particular time, using traditional static and dynamic program analysis techniques. One fundamental problem is their *scalability*. In contrast, our history-based aspect mining approach scales well to industrial-sized projects such as Eclipse with million lines of codes.

Formal concept analysis provides a framework to mine and understand aspect candidates: A transaction is a relation over locations and methods where aspect candidates are the maximal blocks of this relation. These form a lattice of super and sub concepts and can be computed efficiently.

Besides general issues such as performance or ease of use, our future work will concentrate on the following topics:

Measure precision We plan to evaluate our technique by manually investigating the top-ranked aspect candidates to check whether they are actual cross-cutting concerns. The resulting precision will measure the effectiveness of our approach.

Combine several transactions Cross-cutting concerns are frequently introduced within one transaction and extended to new locations in later transactions. Although such concerns are recognised by our technique as several aspect candidates, these candidates may be missed. To locate such aspect candidates, we will use *localities*. For instance, two transactions are related if they changed the same locations or were created by the same developer.

For future and related work regarding history-based aspect mining, see:

`http://www.st.cs.uni-sb.de/softevo/`

7. REFERENCES

[1] S. Breu. Aspect Mining Using Event Traces. Master's thesis, University of Passau, Germany, Mar. 2004.

[2] S. Breu. Extending Dynamic Aspect Mining with Static Information. In *Proceedings of 5th International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 57–65. IEEE Computer Society, Sept./Oct. 2005.

[3] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proceedings of 19th International Conference on Automated Software Engineering (ASE)*, pages 310–315. IEEE Press, Sept. 2004.

[4] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin, 1999.

[5] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, UC, San Diego, 1999.

[6] J. Hannemann and G. Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns*, 2001.

[7] G. Kiczales et. al. Aspect-Oriented Programming. In *Proceedings of 11th European Conf. on Object-Oriented Programming (ECOOP)*, 1997.

[8] J. Krinke and S. Breu. Control-Flow-Graph-Based Aspect Mining. In *1. Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE)*, Nov. 2004.

[9] C. Lindig. Fast concept analysis. In G. Stumme, editor, *Working with Conceptual Structures – Contributions to ICCS 2000*, pages 152–161, Germany, 2000. Shaker Verlag.

[10] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proc. of European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 296–305, New York, NY, USA, 2005. ACM Press.

[11] N. Loughran and A. Rashid. Mining Aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD)*, 2002.

[12] M. Marin, L. Moonen, and A. van Deursen. A classification of crosscutting concerns. In *ICSM*, pages 673–676. IEEE Computer Society, 2005.

[13] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 132–141. IEEE Computer Society, Nov. 2004.

[14] D. Shepherd and L. Pollock. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report 2004-03, U Delaware, 2003.

[15] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE-21*, pages 107–119, 1999.

[16] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 112–121. IEEE Computer Society, Nov. 2004.

[17] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. of Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 97–106. IEEE Computer Society, 2004.

[18] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[19] C. C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proc. of the International Workshop on Mining Software Repositories*, pages 7–11, May 2005.

[20] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland, May 2004.

A Lightweight Approach to Technical Risk Estimation via Probabilistic Impact Analysis

Robert J. Walker, Reid Holmes, Ian Hedgeland
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{rwalker, rtholmes}@cpsc.ucalgary.ca

Puneet Kapur, Andrew Smith
Chartwell Technology Inc.
Calgary, Alberta, Canada
{pkapur, asmith}@chartwelltech.com

ABSTRACT

An evolutionary development approach is increasingly commonplace in industry but presents increased difficulties in risk management, for both technical and organizational reasons. In this context, technical risk is the product of the probability of a technical event and the cost of that event. This paper presents a technique for more objectively assessing and communicating technical risk in an evolutionary development setting that (1) operates atop weakly-estimated knowledge of the changes to be made, (2) analyzes the past change history and current structure of a system to estimate the probability of change propagation, and (3) can be discussed vertically within an organization both with development staff and high-level management. A tool realizing this technique has been developed for the Eclipse IDE.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.7 [**Software Engineering**]: Management.

General Terms: Design, Management, Measurement.

Keywords: Technical risk estimation, decision support, revision history, probabilistic impact analysis.

1. INTRODUCTION

Making good decisions is key to successful development, yet remains a difficult task in an evolutionary development setting. Many organizations struggle to consider both "managerial" and "technical" factors on an objective basis. "Managerial" factors that must be considered include predictions of market forces, conflicting stakeholder interests, and budgetary constraints [23]; "technical" factors include the ease with which proposed extensions can be accommodated by the current software structure [3]. The organizational difficulties arise from the fact that those with the decision making roles typically have the least access to detailed technical knowledge [3], while those with the detailed technical knowledge have the least ability to influence decisions about the direction of development [19]. To bridge this gap, a means for assessing the technical risk of proposed changes is needed that can be audited, for the sake of objectivity, and can serve as the basis for vertical communication within an organization.

Some authors emphasize the risk of introducing flaws into software [12] or causing the failure of a software project [3]. While these are clearly significant threats, we wish to consider the risk of any modification task in a more general sense: the risk of an event is defined as the product of the probability of the event and the cost of the event should it happen. Thus, we can see that even unlikely events with very high costs can result in unacceptably high risks. We expect that an analyst must have at least an approximate sense of key points within a software system that are likely to change. The task then becomes one of determining to what extent these key changes are likely to cause other changes in a cascading sequence: probabilistic change impact analysis. A variety of approaches to change impact analysis have been proposed in the past, but none is appropriate to our context. Many of these techniques expect to be provided with implementations of the initial changes in order to perform their analyses [14]; this is not practicable at an early planning stage. Other change impact techniques exist that try to support decision-making at early phases; some of these expect complete and accurate documentation to be available for analysis [25], others require detailed grammars to be specified as an input to the analysis [13], and still others depend solely on the qualitative judgment of a set of experts [21]. None of these techniques copes well with the assessment and communication of technical risk within an evolutionary development and within an organization in which decision making tends to be separated from detailed, technical knowledge.

Instead, we propose a decision support technique (1) that supports simple entry and update by an analyst of even weakly-estimated knowledge of likely changes, (2) that automatically performs change impact estimation based on such "educated guesses," and (3) that can be used as a basis for vertical communication within an organization. Our technique works from three inputs: a structural dependence graph that is automatically extracted from a software project; change history data for that project that is automatically extracted from a CVS repository; and an indication by an analyst of the key components where a proposed feature is expected to result in a definite change. The technique then uses the structural dependence graph and change history data to estimate the probability that these definite changes will propagate to the rest of the system. Our algorithms are defined in such a way that changing the indication of the definite changes is simple and the results recomputed in real time. The technique has been implemented as a plugin to the Eclipse IDE and deployed to our industrial collaborators for an initial evaluation.

The remainder of the paper is structured as follows. Section 2

MSR'06, May 22–23, 2006, Shanghai, China.

describes our concrete tool implemented as an Eclipse plugin. Section 3 outlines the data extraction steps used to drive the technical risk estimation algorithms. Section 4 describes and analyzes the theoretical model on which our technique is based. A preliminary, informal industrial evaluation has been performed and is described in Section 5. An analysis of the potential weaknesses of this approach, future work, and remaining issues are discussed in Section 6. Section 7 considers related work. The contributions of this paper are a theoretical model for lightweight, probabilistic change impact estimation and a discussion of how this model can be used for decision-support in an industrial context.

2. THE TRE TOOL

Technical risk estimation is a process of specifying starting points for changes and estimating the likely propagation of those changes to the rest of the system. We have implemented a tool, named TRE, to support this process as a plugin to the Eclipse integrated development environment. The TRE tool analyzes structural dependencies between Java files within a project, plus historical data from a CVS repository regarding old versions of those files, to perform its estimations. Analysis is performed at the granularity of types; we consider alternatives in Section 6.

Four steps are needed to perform technical risk estimation: (1) extraction of dependency structure from the project source; (2) extraction of change history data from a CVS repository; (3) creating a conditional probability (CP) graph model (or loading an existing CP graph model); and (4) interacting with the CP graph model to make technical risk estimates. The first three steps are supported via Eclipse wizards; Figure 1 shows an example.

Extraction of dependency structure operates on one or more Eclipse projects residing in the workspace, as specified by the analyst. Currently, dependency analysis is performed only on Java files. The result of this extraction is an XML file representing the dependency graph for the types declared within the project. Dependencies on types external to the project are noted as such. External types are considered immutable for the purposes of technical risk estimation. Our interpretation of "dependency" is explained further in Section 3.1.

Extraction of change history data operates on a module in the CVS repositories that the workspace records (these can be viewed and modified via the standard CVS Repositories view of Eclipse). The result of this extraction is an XML file representing the inferred atomic change sets that have occurred in the repository or repositories specified by the analyst. The inference process performed by the tool is described in Section 3.2.

A CP graph model can be generated from any structural dependency graph and any change history data. The CP graph is effectively a structural dependency graph annotated with probabilities on the dependency edges; each of these represents the conditional probability that a change to the target node will result in a change to the source node. The way in which these conditional probabilities are computed is described in Section 3.3. The CP graph is also written to an XML file; this file may be reloaded at later times to continue analysis.

Finally, the technical risk graph model may be used to estimate the risk of performing proposed changes to a project. The analyst indicates which types declared in the project will be the seed points for change. TRE then estimates the technical risk by propagating these seed points through the remainder of the CP graph according to the conditional probabilities annotating the dependencies. Details of the algorithms used are described in Section 4. The risk of changing a type is defined as the product of the probability of changing that type and the cost of changing that type. The total

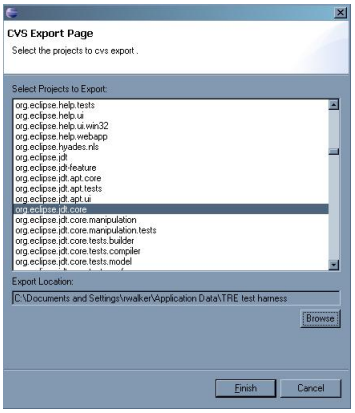


Figure 1: The Eclipse wizard used to select CVS projects for extracting change history data. Here, change history data is about to be extracted from the `org.eclipse.jdt.core` project.

risk of a proposed change is the sum of the risks of changing all the types in the project. Currently, the cost of changing any one type is defined uniformly as 1 for the sake of simplicity. Various alternatives are possible; we consider this issue further in Section 6.

An Eclipse perspective has been implemented to simplify the analyst's interaction with the TRE tool during technical risk estimation. In Figure 2, we see an analysis being performed on a risk graph representing a portion of `org.eclipse.jdt.core`. The perspective provides two views: on the left is the *Risk Graph Nodes* view, and on the right is the *Technical Risk Results* view. Four buttons are present on the tool bar for the Risk Graph Nodes view: *Extract Structure*, *Extract History*, *Create CP Graph*, and *Select CP Graph*. The first three correspond to the first three steps in the process described above; the fourth allows an existing CP graph to be reloaded.

The final, interactive step of the technical risk estimation process proceeds as the analyst selects or unselects the types displayed in the Risk Graph Nodes view. In the example, we see that `org.eclipse.jdt.core.IField` and `org.eclipse.jdt.core.IMethod` have been selected by the analyst. The computation performed by the tool is displayed both in the Technical Risk Results view and on the status line. In the Technical Risk Results view is a list of the types in the project along with the risk of that type changing and the uncertainty in that risk calculation. The types are sorted and coloured according to the calculated risk: the most intense red (resp. darkest grey in a greyscale rendering) and the highest risk is at the top, shading to white and the lowest risk at the bottom. Note that the risk of the selected types changing is currently defined as 1, and so these appear at the top of the list.

3. DATA PREPARATION

Issues involving the design of the basic data extraction and preparation steps of the tool are considered in this section. We begin with structural dependency extraction in Section 3.1, continue with change history data extraction in Section 3.2, and end with the construction of a simple conditional probability model annotating the structural dependencies in Section 3.3.

3.1 Structural Dependency Extraction

To perform structural dependency extraction, TRE begins by re-

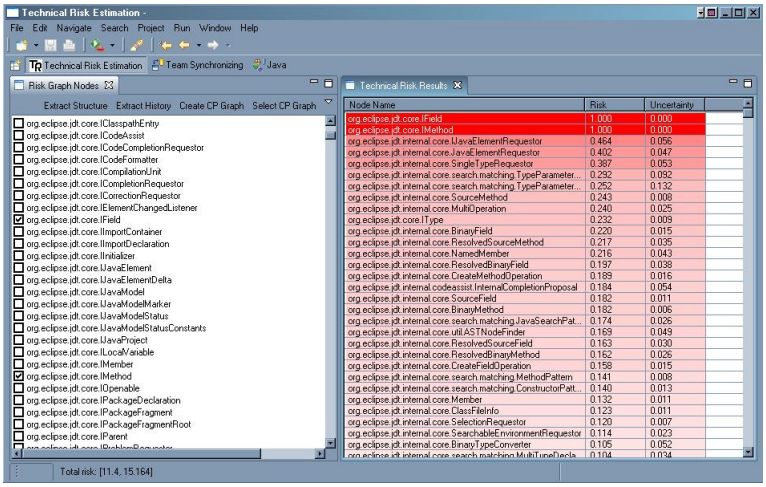


Figure 2: Technical Risk Perspective in Eclipse, showing interaction involving technical risk estimation on JDT core.

questing that an AST be constructed for the selected project(s). A subclass of `org.eclipse.jdt.core.jdom.ASTVisitor` determines dependencies by visiting the nodes in this AST. Type `A` is considered dependent on type `somepkg.B` if and only if: (a) the name `B` (or `somepkg.B`) is explicitly mentioned by `A` and this definitely resolves to type `somepkg.B`; or (b) `A` contains an expression that definitely resolves to type `somepkg.B` and the result of this expression is used within a larger expression, e.g., the result type of a method invocation resolves to `somepkg.B` and this is used as the prefix to another method invocation. Other definitions of dependency could have been chosen; this one was convenient.

A structural dependency graph $S = (T, D)$ is defined as the result of this process, where T is the set of types declared within the selected project(s) and D is the set of dependencies between types as described above. Self-dependencies are ignored. More formally,

$$D \subseteq T \times T \setminus \{(t,t) | t \in T\}. \quad (1)$$

3.2 Change History Extraction

To perform change history extraction, TRE traverses the specified module within a CVS repository. For each file, the log entries are retrieved (instances of `org.eclipse.team.internal.ccvs.core.ILogEntry`), and information on the author, comment, and timestamp stored for the file revision are recorded.

Atomic change sets are inferred by comparing log entry data. Two files that share the same author and comment, and where the timestamps of adjacent check-ins differ by less than three minutes [11], are considered members of the same atomic change set.

The change history is then recorded as an XML file consisting of a sequence of atomic change sets recording the author and comment, and the earliest of the timestamps, plus the set of files that were modified.

3.3 CP Graph Construction

To construct a conditional probability (CP) graph, the structural dependency graph is initially annotated with data from the change history. The number of revisions v_i to each type t_i is recorded at each node in the structural dependency graph. For each edge $e_{i,j} = (t_i, t_j)$ in the graph, the number of times that t_i and t_j occur in the same atomic change set (noted $v_{ij} = v_{ji}$) is recorded at that edge. The conditional probabilities of a change propagating across each edge $e_{i,j}$ may then be calculated as:

$$\Pr(t_i|t_j)|_{e_{i,j}} \approx v_{ij}/v_i. \quad (2)$$

To deal with the discreteness of the data and its occasional poor quality, the results of Equation 2 must be adjusted. This equation makes two assumptions: (1) that $v_i \geq v_{ij}$; and (2) that $v_i > 0$. If either is false, the result will be an invalid probability, so the result must be clamped to fall within the unit interval $[0, 1]$.

To deal with sparse data, we track an interval of conditional probabilities. In the case of few or no revisions, both the numerator and denominator can be 0; nothing can then be said other than that the probability of a change propagating across an edge is somewhere between 0 and 1. Similarly, we wish to account for the difference in quality between evaluating, e.g., $1/10$ and $100/1000$; while each would result in a calculated conditional probability of 0.1, our confidence in the latter would be much stronger. We borrow a simple approach involved in estimating the accuracy of physical measurements: we consider the error in the numerator and denominator to be ± 0.5. The results are again clamped to the unit interval.

Combining these adjustments, we arrive at the following equations:

$$\min_{\Pr}(t_i|t_j)|_{e_{i,j}} = \quad (3)$$
$$\begin{cases} 0 & \text{if } v_i = v_j = 0, \\ \max\left\{0, \frac{v_{ij} - 0.5}{v_i + 0.5}\right\} & \text{otherwise;} \end{cases}$$

$$\max_{\Pr}(t_i|t_j)|_{e_{i,j}} = \quad (4)$$
$$\begin{cases} 1 & \text{if } v_i = 0, \\ \min\left\{1, \frac{v_{ij} + 0.5}{v_i - 0.5}\right\} & \text{otherwise.} \end{cases}$$

(Continuing our simple example, 1/10 would result in a conditional probability range [0.048,0.158] while 100/1000 would result in a conditional probability range [0.099,0.101]).

The conditional probability graph G is then recorded in an XML file as a structural dependency graph $S = (T, D)$ annotated with the conditional probability intervals for each edge, computed according to Equations 3 and 4. More formally,

$$G = (S, \pi_C : D \mapsto \mathcal{I}), \text{ where} \quad (5)$$
$$\mathcal{I} = \{[m,n] | m \geq 0 \wedge n \leq 1 \wedge m \leq n\}. \quad (6)$$

4. THEORETICAL MODEL

In this section, we consider details of the algorithms underlying the TRE tool and their formal basis. The technical risk estimation step proceeds from a set of types marked as seed points for change by the analyst. The probability of these types changing is defined as 1, although the algorithms below could make use of any constant probabilities attached to these seeds.

In Section 4.1, we describe how the probabilistic change impact can be computed from a conditional probability graph and a set of seed types to provide a probabilistic change impact model. In Section 4.2, the final step of estimating the technical risk is considered.

4.1 Probabilistic Change Impact Analysis

Assume that each modification to a type is due either to an immediate modification (i.e., the type is in the seed set) or to a propagation across a sequence of direct dependencies stemming from such an immediate modification.

Begin with a conditional probability graph G as defined in Section 3.3. Let $\Delta_0 \subseteq T$ be a set representing the seed types that the analyst assumes will be immediately modified. We wish to determine the probable change impact to the remainder of the types in the project, i.e., for every type $t \in T$, we wish to determine the probability that t will be modified given that every type in Δ_0 is modified. We can consider this task to involve the construction of a fuzzy set $\Delta = (T, \mu)$ where $\mu : T \mapsto [0, 1]$ is a membership function indicating the probability that each type will change [28].

From the definition of conditional probability we have that

$$\Pr(t_1 \cap t_2 \cap t_3) = \Pr(t_1 | t_2 \cap t_3) \cdot \Pr(t_2 | t_3) \cdot \Pr(t_3)$$

for any three types t_1, t_2, t_3. We have the assumption that a modification to a given t can occur only either because $t \in \Delta_0$ (in which case this probability is 1) or because a type upon which t depends has changed. There must exist a path through G from some type δ in Δ_0 to t for t to have changed. For every path $\delta, t_1, t_2, \ldots, t_n, t$, the probability that t must change is:

$$\Pr\left(t \cap \delta \cap \bigcap_{i=1}^{n} t_i\right) = \Pr\left(t \left| \delta \cap \bigcap_{i=1}^{n} t_i \right.\right) \cdot \Pr\left(t_1 \left| \delta \cap \bigcap_{i=2}^{n} t_i \right.\right) \cdot \Pr\left(t_2 \left| \delta \cap \bigcap_{i=3}^{n} t_i \right.\right) \cdot \ldots \cdot \Pr\left(t_n | \delta\right) \cdot \Pr\left(\delta\right).$$

This equation simplifies significantly because a given type in the path will change only if its predecessor changes; hence, each intersection collapses to the type at the destination of the path:

$$\Pr\left(t\right)|_{\delta} = \Pr\left(t | t_1\right) \cdot \Pi_{i=1}^{n-1} \Pr\left(t_i | t_{i+1}\right) \cdot \Pr\left(t_n | \delta\right) \cdot \Pr\left(\delta\right). \quad (7)$$

There may be more than one path that leads from δ to t, and there may be many types in Δ_0 from which paths lead to t. Each path itself yields a fuzzy set Θ_i indicating the probability that a change to its start will propagate to parts on that path. The fuzzy set Δ that we are interested in determining is simply the union over every Θ_i that is yielded from a path beginning at some element of Δ_0. We consider the probability that t will change to be the maximum of the probabilities calculated along all possible paths from a part in Δ_0 to t, which is consistent with the standard definition of the union of fuzzy sets [28].

We can see that the probability of a change propagating from a source type s to a target type t is analogous to finding the longest path between two vertices in a graph. Because the probability will either remain constant or decrease at each step, infinite paths due to cycles do not cause us difficulties. We proceed with a variation on a modified Dijkstra's algorithm [4]. In this variation, probability is analogous to a reciprocal of distance, requiring the calculation of maxima to replace minima, 0 to replace ∞, etc. In the following algorithm, G is a conditional probability graph as defined in Equation 5, and $s \in T$ is a distinguished source type; the output is a function $\rho_s : T \mapsto \mathcal{I}$. The lower bound $\check{\rho}_s$ and upper bound $\hat{\rho}_s$ of ρ_s for each value of T will be calculated separately. Furthermore, we define $D_t = \{q | (q, t) \in D\}$ be the types directly dependent on type t.

CHANGE-PROBABILITY(G, s)

```
1   for every type t ∈ T
2       ρ̌_s(t) := 1, W := T
3       for q ∈ T \ {t}
4           ρ̌_s(q) := 0
5       while W ≠ ∅
6           find some w ∈ W such that ρ̌_s(w) is maximal
7           W := W \ {w}
8           for v ∈ W ∩ D_w
9               ρ̌_s(v) := max{ρ̌_s(v), ρ̌_s(w) × Pr_min(v|w)}
10  for every type t ∈ T
11      ρ̂_s(t) := 1, W := T
12      for q ∈ T \ {t}
13          ρ̂_s(q) := 0
14      while W ≠ ∅
15          find some w ∈ W such that ρ̂_s(w) is maximal
16          W := W \ {w}
17          for v ∈ W ∩ D_w
18              ρ̂_s(v) := max{ρ̂_s(v), ρ̂_s(w) × Pr_max(v|w)}.
```

The proof that CHANGE-PROBABILITY computes the probability that f will change given that s will change is largely identical to that for Dijkstra's algorithm. The direction of inequalities is reversed, and the sums are replaced with products, which does not alter the argument. Thus, $\rho_s(f)$ represents the maximum product of the input conditional probabilities (for a lower bound traversal or for an upper bound traversal) over any path from s to f. Given this and that $\Pr(s) = 1$, $\rho_s(f) = \Pr(f)|_s$ by Equation 7.

An implementation of Dijkstra's algorithm that uses an unsorted working set has running time in $O(|T|^2)$; a more rational priority queue implementation based on Fibonacci heaps reduces this to $O(|T| \log(|T|) + |D|)$. The changes introduced by CHANGE-PROBABILITY do not alter these arguments.

Because, in general, we will want to know the probability of changing each type given that some set of source types will change, we can consider some alternatives. The choice that we have chosen to implement repeats the computation of CHANGE-PROBABILITY for each of the types $\delta \in \Delta_0$. We compute for each care of δ only on demand, and cache the results. Other alternatives are considered in [26].

We define a *risk graph* R to be a structural dependence graph augmented with the change propagation probability functions ρ_s for all $s \in T$:

$$R = ((T, D), \{\rho_s : T \mapsto \mathcal{I} | s \in T\}). \quad (8)$$

Attempting to access one of these functions that has not been cached results in its computation.

For a given risk graph R and seed set Δ_0, our task is now to determine the fuzzy set (Δ, μ) representing the probability that a change will spread from the seed set. The membership function μ

must also be computed with lower ($\check{\mu}$) and upper ($\hat{\mu}$) bounds, for each type. The algorithm below provides this step.

FUZZY-CHANGE-SET(R, Δ_0)

```
1  Δ := T
2  for every type t ∈ T
3      find some maximal ρ̌_δ(t) ∈ {ρ̌_d(t)|d ∈ Δ_0}
4      μ̌(t) := ρ̌_δ(t)
5      find some maximal ρ̂_δ(t) ∈ {ρ̂_d(t)|d ∈ Δ_0}
6      μ̂(t) := ρ̂_δ(t)
```

The total running time for FUZZY-CHANGE-SET is in $O(|T|^2)$ for an implementation based on unordered sets, or $O(|T|)$ for the Fibonacci heap implementation. Details of this analysis can be found elsewhere [26].

4.2 Estimation of Technical Risk

If we can assume that there exists a cost-of-change function $\kappa : T \mapsto \Re^+$ that is independent of paths through the risk graph, and given the fuzzy set (Δ, μ), we can compute the total technical risk τ of performing a change Δ_0 as:

$$\tau|_{\Delta_0} = \sum_{t \in T} \kappa(t) \cdot \mu(t)|_{\Delta_0}. \quad (9)$$

Ignoring the cost of calculating κ, Equation 9 consists of a simple sum of products. The functions μ and κ can each be recorded in a set sorted in the same order as T, thus yielding $O(1)$ lookup times. The total running time to compute Equation 9 will thus be in $O(|T|)$. As a starting point, we have used the simplistic notion that $\kappa \equiv 1$; we consider more realistic options in Section 6.

5. PRELIMINARY EVALUATION

As a preliminary step in evaluating the efficacy of the approach, a study was undertaken by our industrial partners on the use of the tool to plan change tasks on their codebase. This study is necessarily informal at this stage, as (un)usability issues inherent with a prototype can easily mask effects of (un)usefulness. For the sake of better understanding how our partners would approach the use of the TRE tool, they were given the freedom to conduct the study in a manner that made sense to them.

The tool was assessed by a team of four individuals: one in a technical decision making role with little current knowledge of the fine-grained code structure; one in a technical lead role involving small-scale design and implementation decisions; and two key front-line developers. This team considered a variety of tasks: three code optimizations, three bug fixes, and three changed requirements; specific details of the tasks are hidden to protect intellectual property of our industrial partners. For each task, the team was first asked to give an estimate of the risk involved, in terms of likely number of files that would need modification. They then used the TRE tool to estimate the risk. And finally, the actual number of files that were modified were compared against the estimates. The team was given a written explanation of the operation of the TRE tool, and encouraged to discuss it as a group.

A serious issue was raised by the team: the granularity at which the tool expects its input is too fine-grained for individuals with technical decision making roles but who do not develop code on a daily basis. People in such roles understand the software at an abstract level, often architectural. But such knowledge is difficult to manually translate into a selection of a set of files. Without the ability to formulate a very coarse model, downwards communication within an organization will not be facilitated by the tool. Fortunately, improving this situation can be achieved through better user interface and workflow design that we discuss further in Section 6.

Task	Team-est. risk	Tool-est. risk	Actual change
Optimization 1	5–10	10–17	10
Optimization 2	2–4	2–13	5
Optimization 3	3–5	1	4
Bug fix 1	1–2	1–2	2
Bug fix 2	11	11	14
Bug fix 3	3	1	3
Changed requirements 1	1–2	11–31	2
Changed requirements 2	1–2	1	3
Changed requirements 3	1–2	5–9	2

Table 1: Data reported by the industrial team for their change tasks.

The team summarized their view of the TRE tool thus: "The tool seems to be most useful with certain types of changes. For bug fixes and requirement changes the results using the tool seem reasonably close. However for optimizations it was not very useful and seems to over-estimate the changes required."

The data that they collected and reported is summarized in Table 1. When asked for more detail regarding what these numbers mean, they explained the process that they used: the team-estimated risk represents how many files they considered likely to change, while the tool-estimated risk were how many files had a 50% or greater probability of changing.

In two of the three optimization tasks, the actual change corresponded to the range of technical risk reported; in the third case, our interpretation is that the current structural dependency model did not account for a propagation either due to a subtle relationship (such as those that Ying and colleagues [27] consider) or due to one that a better model could account for, e.g., the need to propagate an interface change through the type hierarchy. For bug fix task 1, the tool appears to have performed well; however, for bug fix tasks 2 and 3, the tool reportedly underestimated the risk. The report for bug fix task 2 may be a data entry error, since the tool in its present form is unlikely not to report a range of risks. For changed requirements tasks 1 and 3, the tool seems to have overestimated the risk; task 2 seems to be suffering from the same structural dependency model issues discussed above.

Despite what seems to objectively be slightly worse performance for the bug fix tasks and much worse for the changed requirements tasks, the team considered the tool more useful for the bug fix tasks and the changed requirements tasks than for the optimization tasks. One conjecture is that the summary data does not fully represent the reality of the situation, e.g., perhaps the correlation of the tool-estimated risks and the actual changes is coincidence and the individual files reported as risky do not correspond to the ones that actually require change. This conjecture must be confirmed or refuted in a future, more controlled study.

The team gravitated towards their natural tendency to threshold the information and treat even a 50–50 chance of change as a prediction of a definite file change and to disregard any lower probabilities; in some circumstances, this will not provide an accurate interpretation of the estimated risk. Also, whenever the change history contains no data, we are able to provide no real information on the probability of change propagation (i.e., 0.5 ± 0.5).

Ultimately, usability issues did limit the evaluation of the usefulness of the approach. Nevertheless, at least three lessons can be taken from this exercise: (1) the detailed risk data needs to be reported in a fashion that takes people's tendency to threshold into

better account, and that deals with uncertainty more clearly; (2) a more iterative approach to building these models and communicating their contents upwards and downwards through an organization is needed, where those with less detailed knowledge can view the data at a more abstract level; and (3) a more comprehensive structural dependency model is needed.

6. DISCUSSION AND FUTURE WORK

A number of extensions and issues remain to be considered and possibly realized in our tool implementation.

6.1 Evaluation of model and tool

Currently, the model and tool we have described make a number of assumptions that may be violated in the real world. Our preliminary evaluation suggests that issues that we had considered to be minor points of usability would actually be too severe for the tool to be adopted in the manner intended. An iterative approach to *in situ* evaluation will continue to be necessary to ensure that additional such points do not intrude in future.

Aside from such practical issues, the question as to whether the model that underlies the tool produces results that are accurate remains unanswered. The TRE tool reports probabilistic predictions that are valid only for a single trial; one should expect that a set of predictions will become very inaccurate after only a few trials. Instead, we have determined a means for correlating the actual profile of a large number of individual probabilistic predictions and single trials with a theoretical profile. In this manner, we hope to vary some of the assumptions of the TRE model, such as the form of the structural dependency model, and compare the results of these variations quantitatively. We are in the process of implementing the infrastructure to collect the data for such evaluations. Details of the theory behind it are left until results from its application can also be reported.

6.2 Sources of error

A number of issues in the way we collect and model data are potential sources of error within our analyses.

An often reported issue is that of the presence of transformations within the codebase, ranging from the renaming of types to large-scale refactorings. We are in the process of applying the technique of origin analysis [7] as a means to better combine data arising before and after such transformations. A finer-grained representation of structural dependencies combined with origin analysis is likely to improve the accuracy of our predictions.

Such transformations are but one impediment to the larger issue of inferring causality that our model ultimately involves. While one can determine atomic change sets either because the repository system supports them explicitly or they can be inferred, approaches (including ours) that depend on atomic change sets as the basis for causality inference are at the mercy of the repository commit style applied by the developers of a given project. For example, are entire feature sets committed at once or are individual files checked in on a rather ad hoc basis? Instead, a stronger model for recovering causality is needed. Utilizing data from change request repositories to bridge the gaps between atomic change sets is one possibility (e.g., [6, 22]).

Of particular interest to our industrial partners is the ability to cope with software that involves multiple languages. Our current approach for extracting structural dependencies would need to support specific combinations of languages to understand how they interact. If the number of languages to be simultaneously supported is small and invariant, the brute force method of providing syntactic and semantic analysis support is likely practicable. However, in situations where configuration details effectively result in a proliferation of small, special purpose languages, the cost of providing such support becomes prohibitive. Lightweight approaches involving lexical analysis (e.g., [15]) must be considered as a more practicable solution.

Our current assumption that the cost of changing any given file is constant (and "1") regardless of any other factors is clearly unrealistic, but serves as a simple starting point. Utilizing data on the size of changes or the error rates connected with previous changes (e.g., [12, 22, 17]) are more sophisticated approaches that we will investigate in future.

6.3 Workflow and organizational issues

It is clear that the input required by the tool at present is too fine-grained to support its use by people with only abstract knowledge of the system under study. On the other hand, an even finer-grained input could be useful where more specific information is known. The theoretical model that we have described in this work does not require that seed points be specified at the granularity of types. Coarser-grained input could be supported by providing a tree-based view that collapses types into packages, for example, or that otherwise provides some form of architectural clustering (e.g., reflexion models [16]). Likewise, the change history data could be analyzed at a finer-granularity to allow input at the method- and field-level when desired.

The issue then becomes one of how to interpret seed markings at coarse granularities in terms of the underlying fine-granularity model. Two options would seem worth pursuing. Marking a coarse-grained item (such as a package) could be treated as equivalent to marking each of its individual constituent types (or methods). This could interact with organizational workflow in such a way that the technical decision maker's initial estimates of risk would be consistently higher than those of the front-line developers. What the social and organizational implications of such a phenomenon might be remains unclear. Alternatively, a fraction of the probability could be assigned to each of the underlying fine-grained items; note that Equation 7 remains valid for probabilities of seed points other than 1.

Other issues of practical importance include the ability to collect the data incrementally as changes happen over time, better user interface controls to adjust the way the results are displayed (e.g., sliders for adjusting thresholds of interest and of pessimism), basic search functionality in the user interface, and the ability to mark nodes with probabilities other than 1 (e.g., to specify "this is guaranteed not to change"). Differentiating the kind of change would also allow the model to propagate different kinds of change in different fashion. The details of such categorizations, the effect on the theoretical model, and the practicality of asking for additional input need to be considered further.

7. RELATED WORK

Various previous work has considered the meaning of "technical risk" and how it should be managed. Technical risk has been seen as a serious factor in industrial development for decades (e.g., [5]), where improved tools and process were often seen as the key means to mitigate it. Initially, technical risk was often equated with the risk that the proposed technology to be used in a development effort would not actually support the application, and so that development effort would fail (e.g., [2, 3]). The move towards separation of managerial decisions and technical decisions has been seen as emphasizing the organizational gulf between decision makers and technical personnel [3]. More recent work attempts to provide questionnaires to guide assessment of risk (e.g., [9]) or to leverage

and combine the opinions of experts [21]. The TRE tool is seen as providing input to the release planning process [21].

Various authors have considered which risk factors are most serious in software development, often based on surveys of project managers' opinions [20, 23]. Others have attempted more objective, quantitative approaches [18]. Still others provide survey based approaches to evaluate the risk in a given project

A large body of work emphasizes the risk of introducing defects into software as it is modified. Belady and Lehman provide a very early attempt at doing this quantitatively [1]. Mockus and Weiss provide a probabilistic model of software failures due to a variety of factors including developer experience and change propagation [12]. A large body of work by Zeller and colleagues focusses on failure prediction based on historical data (e.g., [22, 17]).

Dependence analysis worked from a strong theoretic basis [8], and has long been seen as important in software modification [10]. Formal undecidability lead to approximate change impact analysis techniques [14]. However, all these techniques require detailed implementations as input. In contrast, Turver and colleagues define a technique for early change impact analysis [25]; unfortunately, it assumes complete and accurate documentation that is rarely available in the development setting that we consider. Moonen has proposed a lightweight impact analysis technique based on the creation of island grammars to describe the syntactic cues that an analyst seeks [13]; however, this approach remains too detailed for the needs of early decision support.

Many approaches try to evaluate the quality of designs based on quantitative measures. Tsantalis and colleagues most recently consider how such quantitative approaches can predict change [24].

Most significant for this workshop are the large number of approaches that use historical data to guide change tasks. Both Ying and colleagues [27] and Zimmermann and colleagues [29] consider how frequent patterns in past change propagation can be used to guide developers in propagating changes that they might otherwise miss. They use thresholding to filter suggestions that are unlikely to be important; in contrast, we must consider the presence of low probability/high cost events in evaluating risk. The approach we have presented in this paper is largely complementary to these, lying at the opposite end of the spectrum between coarse-grained planning and fine-grained implementation.

8. CONCLUSION

We have described a theoretical model for technical risk estimation and its concrete realization as the TRE tool. TRE has been designed to help organizations make more informed decisions about the risks associated with modifying software elements within their changing systems. By providing an abstract view about the potential costs of a particular change to managers, and more concrete data to developers, the tool facilitates communication between these two groups. Through supporting the decision making process, TRE allows organizations to ground their development plans both in the structural nature of their source code and the past development history that the system has undergone.

Although preliminary evaluation has shown a need to improve the tool in specific ways to support this vertical communication, we believe that most of these issues can be addressed by small improvements in workflow-support and usability.

Ultimately we see the TRE tool as being complementary to many of the approaches that have come out of the repository mining community in recent years. Our work extends these approaches to provide an objective foundation for making decisions at various levels of industrial organizations about the technical risk of software modifications.

9. ACKNOWLEDGMENTS

We wish to thank Stefania Bertazzon for comments on an early draft of this paper, and the anonymous reviewers for their efforts. This work was supported in part by NSERC and in part by Chartwell Technology Inc.

10. REFERENCES

[1] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems J.*, 15(3):225–252, 1976.

[2] B. I. Blum. Three paradigms for developing information systems. *Proc. Int'l Conf. Softw. Eng.*, pages 534–543, 1984.

[3] C. Chittister and Y. Y. Haimes. Assessment and management of software technical risk. *IEEE Trans. Systems, Man and Cybernetics*, 24(2):187–202, 1994.

[4] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:169–271, 1959.

[5] H. Fischer. Computer system simulation of an on-line interactive command and control system. In *Proc. Winter Simulation Conf.*, pages 333–340, 1971.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int'l Conf. Softw. Maintenance*, pages 23–32, 2003.

[7] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Sys.*, 12(1):26–60, Jan. 1990.

[9] J. Kontio, G. Getto, and D. Landes. Experiences in improving risk management processes using the concepts of the Riskit method. In *Proc. ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pages 163–174, 1998.

[10] J. P. Loyall and S. A. Mathisen. Using dependence analysis to support the software maintenance process. In *Proc. Conf. Softw. Maintenance*, pages 282–291, 1993.

[11] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Method.*, 11(3):1–38, 2002.

[12] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical J.*, 5(2):169–180, 2000.

[13] L. Moonen. Lightweight impact analysis using island grammars. In *Proc. Int'l Wkshp. Program Comprehension*, pages 219–228, 2002.

[14] M. Moriconi and T. C. Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Trans. Softw. Eng.*, 16(9):980–992, 1990.

[15] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Method.*, 5(3):262–292, 1996.

[16] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. ACM SIGSOFT Symp. Foundations Softw. Eng.*, pages 18–28, 1995.

[17] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. Int'l Conf. Softw. Eng.*, 2006. To appear.

[18] D. E. Neumann. An enhanced neural network technique for software risk analysis. *IEEE Trans. Softw. Eng.*, 28(9):904–912, 2002.

[19] K. S. Rajeswari and R. N. Anantharaman. Development of an instrument to measure stress among software professionals: Factor analytic study. In *Proc. SIGMIS Conf. Computer Personnel Research*, pages 34–43, 2003.

[20] J. Ropponen and K. Lyytinen. Components of software development risk: How to address them? A project manager survey. *IEEE Trans. Softw. Eng.*, 26(2):98–112, 2000.

[21] O. Saliu and G. Ruhe. Software release planning for evolving systems. *Innovations in Systems and Softw. Eng.*, 1(2), 2005. To appear.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Wkshp. Mining Software Repositories*, pages 24–28, 2005.

[23] A. Tiwana and M. Keil. The one-minute risk assessment tool. *Commun. ACM*, 47(11):73–77, 2004.

[24] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Trans. Softw. Eng.*, 31(7):601–614, 2005.

[25] R. J. Turver and M. Munro. An early impact analysis technique for software maintenance. *J. Softw. Maintenance: Res. and Pract.*, 6:35–52, 1994.

[26] R. J. Walker, R. Holmes, I. Hedgeland, P. Kapur, and A. Smith. A lightweight approach to technical risk estimation via probabilistic impact analysis. Tech. rep. 2006-817-10, Computer Science, Univ. of Calgary, 2006.

[27] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[28] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[29] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.

Fine Grained Indexing of Software Repositories to Support Impact Analysis

Gerardo Canfora
Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
canfora@unisannio.it

Luigi Cerulo
Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
lcerulo@unisannio.it

ABSTRACT

Versioned and bug-tracked software systems provide a huge amount of historical data regarding source code changes and issues management. In this paper we deal with impact analysis of a change request and show that data stored in software repositories are a good descriptor on how past change requests have been resolved. A fine grained analysis method of software repositories is used to index code at different levels of granularity, such as lines of code and source files, with free text contained in software repositories. The method exploits information retrieval algorithms to link the change request description and code entities impacted by similar past change requests. We evaluate such approach on a set of three open-source projects.

Categories and Subject Descriptors

H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement, Experimentation

Keywords

Mining Software Repositories, Impact Analysis

1. INTRODUCTION

CVS and Bugzilla are two tools for configuration management used with success by the open source community for sharing knowledge. The quality of data, in particular free text, such as bug comments, bug descriptions, and feature proposal definitions, is a critical need in an environment in which no people meetings, no phone calls, and no coffee break discussions are possible [8]. This leads to consider such software repositories interesting data sources, useful for developing text mining techniques to assist project managers and developers in their maintenance activities.

Natural language is widely used in many software engineering artifacts and it is not unusual to find in the literature models based on text mining techniques, information retrieval algorithms, and natural language processing approaches. In [1] a probabilistic information retrieval model has been used to map source code artifacts with documentation. In [14], and [19] text mining techniques have been used in free text contained in software repositories for mining, respectively, concept keyword, and project information.

In this paper we take advantage of free text stored in software repositories to build a textual representation of code entities at different levels of granularity, such as lines of code and source files. This can help in the problem of impact analysis, that is the identification of the work products affected by a proposed Change Request (CR), either a bug fix or a new feature request. Developers can know what are the code entities he/she should work on to resolve a given change request. Project managers can have an estimation of what are the impacted code entities in the next release, useful to focus testing effort.

The method has been introduced in [3] considering a level of granularity restricted to source files. In a set of four case studies, we obtained a precision that ranges between 30% and 78%. In this paper we consider a finer level of granularity, lines of code, and we show that this introduces an improvement at least of 10%, at the cost of spending more time and space for the index.

The paper is organized as follows: next section provides an overview about related work in the field of impact analysis; section three describes the bug resolution process generally adopted by the open source community and what are the free text left by developers; section four introduces the concept of line history table showing how changes at line-of-code level can be recovered from a CVS repository; section five introduces the approach of impact analysis; section six shows the application and validation of the approach in three case studies; the final section concludes the paper with open issues and future works.

2. RELATED WORK

Traditionally, impact analysis has been faced by static analyzing the product [2]. Many approaches are based on traceability analysis and dependence analysis. Traceability analysis identifies affected software entities using explicit traceability relationship. Some methods use a traceability matrix to represent relationships and the impacted objects

MSR'06, May 22–23, 2006, Shanghai, China.

are inferred by computing the transitive closure of the matrix. Dependence analysis attempts to assess the effects of a change based on semantic dependencies between program entities; a technique is to use static and/or dynamic slicing [11]. Expert judgement and code inspection are also used; however, expert predictions have been shown to be frequently incorrect [12], and source code inspection can be prohibitively expensive [15]. The availability of data on software process, such as those deriving from software and change repositories, can provide new opportunities for impact analysis. In particular, approaches to predict the impact and propagation of changes can be found in [20, 18]. They use heuristics, such as historical co-change and co-authorship, to derive the set of impacted source file or program entities. A method to evaluate the performance of change propagation heuristics has been introduced in [9]. These methods predict the impacted files starting from a given initial source file, while the approach we present in this papers starts from a change request description.

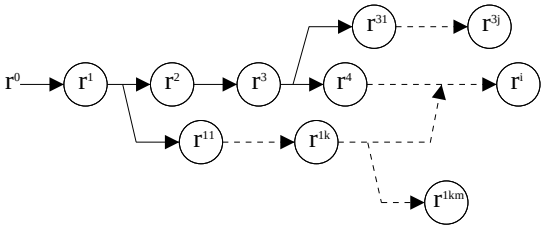


Figure 1: Source file revisions graph

Table 1: Line history table

r^1	...	$r^{ijk...l-1}$	$r^{ijk...l}$	line #
	...			1
	...			2
...	...	...	...	...
	...			n

3. FREE TEXT IN SOFTWARE REPOSITORIES

The resolution of a new CR, in many open source projects tracked by Bugzilla and CVS, usually follows a very simple workflow. A reporter proposes a new CR that can be a bug he/she discovered or an enhancement feature he/she likes to suggest. The CR is stored in the new CR database, after a validation performed by the maintainer of the project to confirm it exists. A developer that has dealt with similar CRs in the past has a wide knowledge of the source code involved and can easily locate the code entities that should be changed. Otherwise, if he/she does not have such knowledge, it is usual to ask for the help of other developers that have resolved similar problems in the past. It is not rare to find in the discussion comments of a CR, topics regarding similar past behaviors resolved in other CRs. An *assigned to* relationship links the developer with the CR. The resolution of the CR evolves to a fixed CR with a commit, in the CVS database, of the source code changes that resolves the CR (*impact* relationship). This relation does not exist in Bugzilla database but, as suggested in [6], it can be derived because usually developers keep track of the impacted files by inserting in the CVS commit comment the id number of the CR. A *resolved by* relationship links the developer, author of the resolution change, and the fixed CR.

This process involves a lot of information both structured and not structured, e.g. composed of free text. A file revision is composed by a set of fields: *revision*, is a number that increases when new changes are committed by the developer; *date*, is the date and time of check-in; *author*, is an identificator of the person who did the check-in; *state* assumes one of the following values: 'exp' means experimental and 'dead' means that the file has been removed; *lines*, the number of lines added and deleted with respect to the previous version of the file; and a final block of free text that contains informal data entered by the developer during the check-in process.

A CR is in many cases represented in XML and it is enclosed generally in a *bug* or *issue* tag containing: *bug-id*, a unique identifier assigned by Bugzilla; *creation-ts*, the date and time of CR creation; *short-desc*, a short description; *product*, the product name; *component*, the component of the system; *reporter*, who has submitted the CR; *assigned-to*, who was assigned the CR for resolution; and *long-desc*, a structure comprising a long description of the CR, *thetext*, who submitted it, *who*, and when, *bug-when*.

4. FINE GRAINED ANALYSIS OF CVS REPOSITORIES

CVS handles revisions of textual files by storing the difference between subsequent revisions in a repository. It provides only information on files and differences, but not which code entities have been changed. For an analysis of fine-grained entities, another preprocessing step is required: each revision is compared with its predecessor and the changes are mapped to entities. In [21] syntactic entities, such as functions, methods, and variable declarations have been considered. In this paper we refer to lines of code entities and recover the history of source code modification in terms of lines that have been added, changed, and deleted during the evolution of a source file. In doing that we will introduce the concept of *line history table* as a tool to visualize source file evolution at the line-level granularity. A similar concept has been introduced in [4] for different purpose.

Figure 1 shows a typical source file revisions graph with different development trunks. Each revision is identified with a sequence of numbers positions: $ijk...l$, in which the last, l, is incremented by one every time a new revision is committed. Revision r^0 represents the empty file. A development trunk can be started from every revision and at some point it can be merged to the trunk from which it derives. When a new development trunk starts the revision identifier is incremented with another number position at the end, initially equal to 1.

Each revision is compared with its predecessor by using the *diff* tool. If a revision is a merge of multiple predecessors, it should get a special treatment, while a revision with no predecessors is compared against an empty file.

A line history table depicts a particular revision $r^{ijk...l}$ and contains a row for each of its source line, and a number of column for all previous revisions belonging to the path that reaches the revision r^0 (table 1). For example, the line history table of revision r^{31} contains the columns: r^1, r^2, r^3, and r^{31}. If merges will not be considered for each

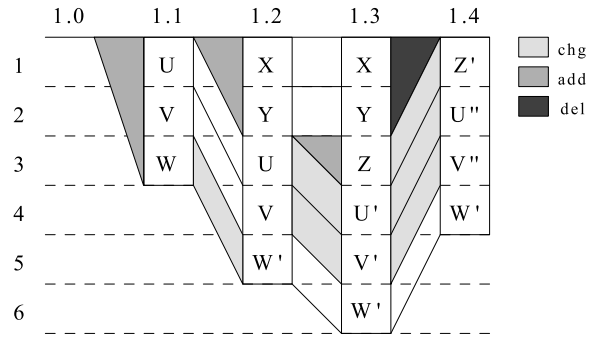


Figure 2: Revisions example

revision, only one line history table can exist. The i-*th* row of the table shows the history of the i-*th* line of the revision by using a marker, 'a' or 'c', in the column corresponding to the revision the line has been respectively added or changed. Some constrains hold for the relative position of markers:

- a line is added to revision only once, then each row contains one, and only one 'a';
- a line can be changed only if it has been added in a previous revision, then in each row, 'a' precedes each changes 'c'.

A line history table can be built from the output of a *diff* command. When comparing two files, *diff* finds sequences of lines common to both files, interspersed with groups of differing lines called *hunks* [13]. There are many ways to match up lines between two given files. The algorithm inside *diff* tries to minimize the total hunk size by finding large sequences of common lines interspersed with small hunks of differing lines.

A diff command is performed between two revisions, called the right and the left revision and the output of a diff command is a sequence of tuples x, yTw, z, in which x, y and w, z are two line number intervals of respectively the left and right revision, and T can be: a, c, or d which means respectively that the left interval has been added, changed, or deleted in the right interval. The column r^i of a line history table of revision r^{i+k} is computed from the output of a diff command performed between revisions r^{i-1} and r^i. Each right interval is translated to line numbers of revision r^{i+k} by examining what happens to this interval in each subsequent revision r^{i+j} for $j = i + 1, ..., k$.

Table 2 shows the output of a *diff* command for four revisions performed on a text file depicted in figure 2. The first revision (1.1) contains three lines. The second revision (1.2) adds two more lines on the top and changes the third line of the previous revision. The third revision (1.3) adds a new line in third position and changes the two lines that follows. The last revision (1.4) deletes the first two lines and changes the three lines that follows. Revision 1.0 means the empty file.

Table 4 shows the line history table of revision 1.4. Diff information between revision 1.0 and 1.1 shows an add of the first three lines (`0a1,3`). The add range will change if we look what happens in subsequent revisions 1.2, 1.3, and 1.4. In revision 1.2, as two more lines are added to the top (`0a1,2`), the range is shifted from 1,3 to 3,5. In revision 1.3, as one line is inserted in position 3 (`2a3`), a shift of size one places the lines in positions 4 to 6. In revision 1.4 two lines have been deleted and the final line positions are shifted up in the range 2,4 as shown in the first column of table 4. Other columns are computed in a similar way by shifting up and down line positions in order to take into account line additions and deletions in subsequent revisions. In this way, we obtain the diff information for each revision translated to line positions relative to the reference revision. Line changes are slightly different if the reference revision is 1.3, as shown in table 3.

In this example, for the purpose of simplicity, we have not considered cases in which the subsequent add and delete are inside the range to be transformed, nor the case in which changes have different left and right range sizes. They can be modeled by considering a range split in the first case and by transforming the change in an add/del + change operation in the second case. The add/del are computed in order to have change subpart of the same size for left and right ranges.

A line history table can be used in a number of ways. Given a system release, the line history table of each revisions belonging to that release can be computed. As an example, for each source line, the number of past revisions until its last change is an indicator of its age, while the number of changes explains its stability.

In the context of this paper we are interested to use the line history table of the current system release, and represent each line of code with free text related to revisions in which the line has been added, or changed. This comprises revision comments and short and long descriptions of CRs that impact those revisions.

Table 2: Diff information between revisions

rev1	rev2	diff
1.0	1.1	`0a1,3`
1.1	1.2	`0a1,2` `3c5`
1.2	1.3	`2a3` `3,4c4,5`
1.3	1.4	`1,2d0` `3,5c1,3`

Table 3: Line history table of revision 1.3

1.1	1.2	1.3	line #
	a		1
	a		2
		a	3
a		c	4
a		c	5
a	c		6

Table 4: Line history table of revision 1.4

1.1	1.2	1.3	1.4	line #
		a	c	1
a		c	c	2
a		c	c	3
a	c			4

5. IMPACT ANALYSIS APPROACH

Our approach to impact analysis is shown in figure 3. A descriptor builder process links free text contained in software repositories with source code entities and an indexing process generates the index used by an information retrieval algorithm to retrieve the ranked list of code entities impacted by a new CR. The hypothesis is that revision comments and impacted CRs are a good descriptor of code entities, such as source files and lines of code, to support impact analysis of new CRs. This is granted by the fact that CVS and Bugzilla are extensively used as tools for knowledge sharing during the software development process with textual data of acceptable quality. We use textual similarity to compute the similarity between a new CR descriptor (i.e. *short-desc*, or *short-desc* + *long-desc*) and the set of source code entities descriptors. The most similar code entities are retrieved and presented to the developer as a first ranked list of probable impacted code entities from which change propagation can start. Textual similarity is a critical part of our approach. The Information Retrieval community dealt with text similarity for a long time. Given a set of text documents and a user information needs represented as a set of words, or more generally as free text, the information retrieval problem is to retrieve all documents relevant to the user [17]. In information retrieval, queries and documents are described by a set of index terms. Let $T = \{t_1, t_2, \ldots, t_n\}$ denotes the set of term used in the collection of documents. Both a document d and a query q are represented as a vector $(x_1, x_2, \ldots, x_n)$ with $x_i = 1$ if t_i belong to the document/query and $x_i = 0$ otherwise. In our approach we have used a probabilistic information retrieval model in which the relevance of a document with respect to a query is computed by evaluating $P(R|d, q)$, that is the probability that a given document d is relevant to a given query q. Different probabilistic models have been proposed in literature to evaluate this probability. We have used the model introduced in [10]. It assumes that each term is associated with a topic, and that a document may be about the topic, or not. Statistic measures about the term occurrences in documents are used to estimate the probability. In particular, a document d is scored with respect to a query q by using the following scoring function:

$$S(d, q) = \sum_{t \in q} W(t) \tag{1}$$

that sums the weight of each query term with respect to the document d on the basis of a weighting function W. An overview of weighting functions can be found in [5]. We have used the following [10]:

$$W(t) = \frac{TF_t(k_1 + 1)}{k_1\left((1-b) + b\frac{DL}{AVDL}\right) + TF_t} \log \frac{N}{ND_t}$$

where TF_t is the frequency of term t in the document, DL is the document length (i.e. number of terms), $AVDL$ is the average document length in the collection, N is the number of documents in the collection, and ND_t is the number of documents in which the term t appears. The constant k_1 determines how much the weight reacts to increasing TF_t, and b is a normalization factor. We have used the values of $k_1 = 1.2$ and $b = 0.75$, which are recommended values for generic English text.

The vector representation is built through an indexing process composed by a number of standard steps usually performed to improve the retrieval performance. The first step, *term tokenizer*, regards the subdivision of free text in a sequence of index terms. A token is a sequence of alphanumeric characters separated by non-alphanumeric characters. In our case we have discarded tokens consisting only of digits. The second step, *stemmer*, serves to lead a term to its root. For example verb conjugation is led to the infinitive verb, plural is led to singular, and so on. Terms are stemmed in order to collapse terms with the same meaning into a single term. In our case we have used the Porter stemmer algorithm for English [16]. The third step, *stopper*, serves as a filter of common words that are not discriminant for the document. We have used a common stop word vocabulary used in the context of English text retrieval enriched with a set of words picked up from the software system domain. For example, we have discarded words such as *bug*, *feature*, and words related to the system under consideration such as *argouml*, *gedit*, and so on. The set of so obtained terms are counted for each document and stored within the document identifier in two data structures, namely direct and inverted indexes. The first stores term occurrences within a document, while the second stores term occurrence among the collection.

In the next two subsections we consider the descriptor building process of code entities at two different levels of granularity, source files and lines of code, and in the next section we show that indexing finer grained code entities gives, in most cases, a better performance in retrieving the impacted source files. Moreover, fine grained indexing can give more rich information as a result because, within the source file, the set of impacted lines of code is also returned.

5.1 File indexing and file retrieval

Source files indexing is performed on descriptors built for each source file belonging to a system release. A source file descriptor, $D(sf)$, is defined as:

$$D(sf) = \sum_{cr \in impact(sf)} D(cr) + \sum_{r \in revision(sf)} D(r)$$

Where $D(cr)$ is the descriptor of the change request cr that impact the source file sf obtained by the concatenation of its short and long descriptions; $D(r)$ is the descriptor of the revision r of the source file sf obtained from its commit comment; and + is the operator of string concatenation.

Source file retrieval is performed by computing the score value of each source file sf with respect to the new CR descriptor by using the equation 1, $S(sf, CR)$. The set of source files in descending order with the score value is returned to the user. Source files with a score value less than a threshold constant t are discarded, usually $t = 0$.

5.2 Line of code indexing and file retrieval

Lines of code indexing is performed on descriptors built for each line of code belonging to each source file of a system release. A line of code descriptor, $D(lc)$, is defined as:

$$D(lc) = \sum_{cr \in impact(lc)} D(cr) + \sum_{r \in revision(lc)} D(r)$$

Where $D(cr)$ is the descriptor of the change request cr

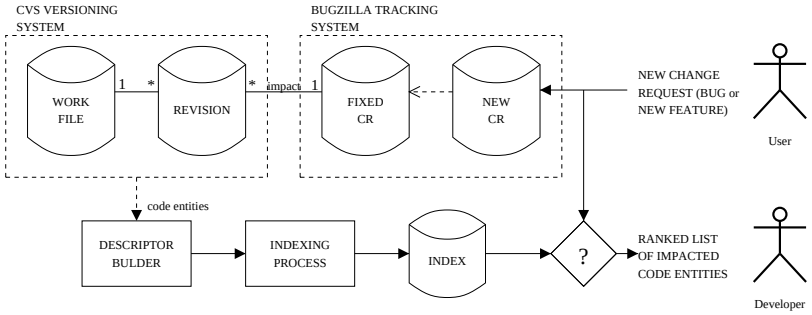


Figure 3: Impact analysis process

that impact the line of code lc; $D(r)$ is the descriptor of the revision r in which the line of code lc has been added or changed; and $+$ is the operator of string concatenation. While $impact(sf)$ and $revision(sf)$ can be derived directly from software repositories, those relative to a line of code, $impact(lc)$ and $revision(lc)$, can be derived by using the line history table of the current system release and considering only those revisions in which the line has been added or changed.

Since, $impact(lc) \subseteq impact(file(lc))$, and $revision(lc) \subseteq revision(file(lc))$, then $D(lc) \subseteq D(file(lc))$, where $file(lc)$ is the source file lc belongs to. When indexing the line-of-code level, the score value of each line, lc, with respect to the new CR descriptor is computed by equation 1, $S(lc, CR)$. We score a source file, sf, by computing the maximum score of lines belonging to it.

$$S(sf, CR) = \underset{lc \in sf}{MAX} \left(S(lc, CR) \right)$$

Other score functions can be defined but this one has given good results. Lines of code indexing is more expensive, in space and time, than source file indexing, by a factor depending on the average length of source files.

5.3 Tool support

We have developed an Eclipse plug-in, named Jimpa, in order to support both indexing and source file retrieval. All steps are completely automated, including the download from the CVS and Bugzilla repositories. As shown in figure 4, the user can write a short explanation of the impacted source files he/she wants to search for. The user can choose the index to use, either fine or coarse grained, respectively lines of code and source files. The search is performed among the current project and the set of source files, ranked by their relevance with change request description, is returned by the search engine and shown in the bottom. The list shows, for each source file, the project relative path location and the relevance weight for the change request description. For fine grained index, the set of finer code entities, such as ranges of impacted lines of code, are shown within the source file. The tool provides the support for setting information retrieval properties such as stop word list, stemmer algorithm, and fields to be included or excluded from the indexing process. Moreover, parameters to access a Bugzilla site can be set in the preference dialog of each Eclipse project. Jimpa runs under Eclipse 3.1 and is hosted on an Eclipse update site at the following URL: http://cise.rcost.unisannio.it/updates/.

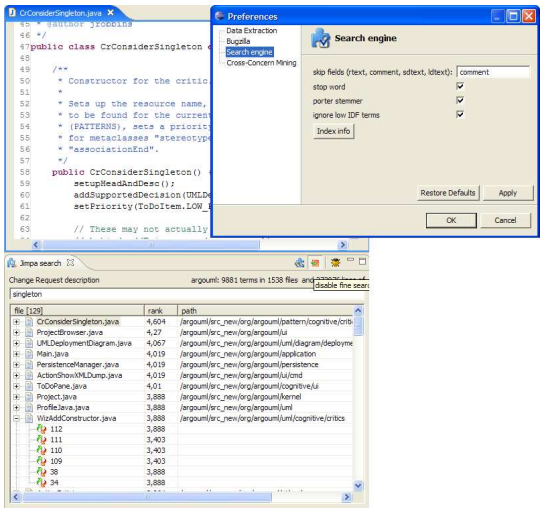


Figure 4: Tool snapshot

Table 5: Open-Source projects

project	files	lines	lines/files	fixed CRs	age
Gedit	117	47913	409.5	116	9 years
ArgoUML	1538	272076	176.9	670	7 years
Firefox	89	42580	467.4	591	4 years

6. CASE STUDY

We have applied the impact analysis approach in three case studies with different characteristics (Tab. 5). The first, Gedit, is a general purpose text editor of the GNOME desktop environment written in C. The second, ArgoUML, is an UML modeling tool written in Java. The third, Firefox, is an Internet browser written in C++.

The results have been assessed using two widely accepted information retrieval metrics, namely, *Precision* and *Recall* [17]. *Precision* is the ratio between the number of relevant documents retrieved for a given query and the total number of documents retrieved for that query. *Recall* is the ratio between the number of relevant documents retrieved for a given query and the total number of relevant documents for that query. In our case study recall and precision indicates how many of the right impacted files have been correctly predicted (recall) and how many of the predicted impacted files are right (precision). We use the same methodology used for evaluating an information retrieval algorithm, that

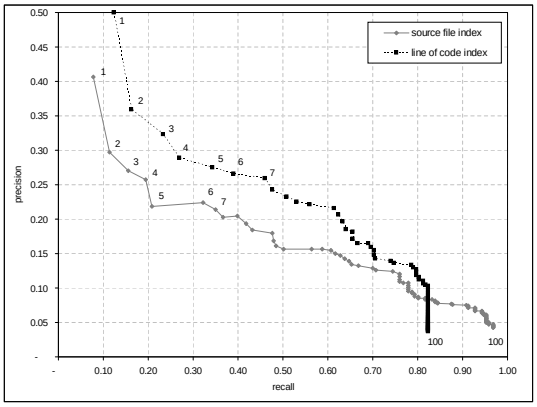


Figure 5: Gedit results

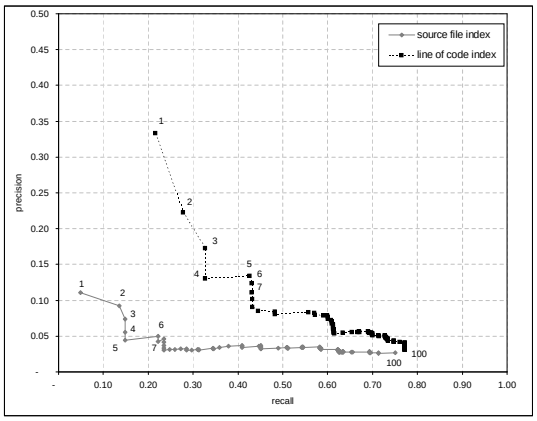


Figure 6: ArgoUML results

is, the retrieved ranked list of documents is considered at different cut levels [17]. A cut level N is the list of the first N ranked documents. For each cut level the behavior of precision and recall is analyzed and traced on a graph.

We have conducted the evaluation by using the leave-one-out assessment technique [7]. For a given CR we have predicted the set of impacted files by using an index without descriptors regarding that CR. The predicted set of files is then compared with the oracle set, that is the files actually impacted by that CR, recovered by considering the presence of the Buzilla id number in the revision comments of the files [6]. We think this is a good oracle as CRs managed in Bugzilla follow basically some accepted guidelines, and one of these is to indicate in the check-in comment the Bugzilla id that identifies the relevant CR.

The performance has been evaluated by using both source file and line of code indexes. Figures 5, 6, and 7 show the results for each of the three systems considered. The curves have been traced by observing the top 100 files ranked by the scoring function and averaging the precision and recall on the number of CRs considered for each system. Each figure contains two curves, relative to source file index and line of code index. A curve contains 100 points, one for each cut level starting from 1 to 100. The first set of points should be read as a measure of overall precision, while the last set of points as a measure of overall recall.

Indexing lines of code produces improvements ranging be-

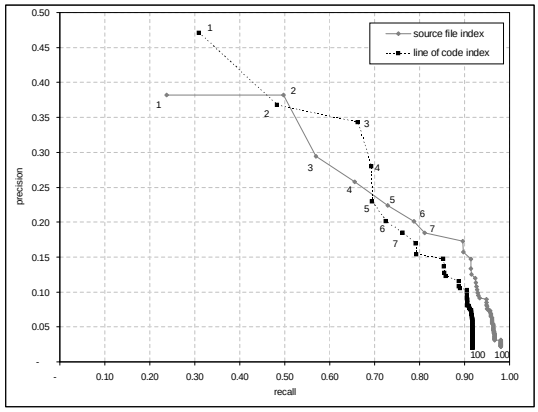


Figure 7: Firefox results

Table 6: Time and space needed to build the indexes

project	source file index		line of code index	
	time	space	time	space
Gedit	2 *sec*	360 *KB*	59 *sec*	8.5 *MB*
ArgoUML	19 *sec*	1.82 *MB*	390 *sec*	52.5 *MB*
Firefox	3 *sec*	300 *KB*	63 *sec*	3.3 *MB*

tween 10% and 20% of top 1 precision for each case considered. Top 100 recall is better for source file index in two cases, ArgoUML and Firefox. An evident improvement is reported for ArgoUML for which the top 1 precision is almost 20% better for lines of code index than for source files index. Is this related to the fact that ArgoUML is written in Java?

Why different performance behaviors occurs for different systems needs to be further investigated. For sure it depends on how software repositories are used in software projects. Usually, projects share a common usage practice driven by the configuration management system but with a some slightly deviation driven by the members of the project.

Table 6 shows the time and the disk space needed to build both source file and line of code indexes for each case considered. Data explains that the increment of the cost, in terms of time and space required to index lines of code, grows percentually more than the increment of performance gained. However this is not a drawback at all because the indexing process takes place only one time to set-up the environment and successive index updates are performed incrementally. Regarding file retrieval response time there is a no evident cost increment as shown in table 7.

7. CONCLUDING REMARKS

Software and change repositories give new opportunities to support the software development process. In this paper

Table 7: Average file retrieval response time

project	source file index	line of code index
Gedit	16.2 *msec*	31.1 *msec*
ArgoUML	266.3 *msec*	375.5 *msec*
Firefox	15.1 *msec*	30.4 *msec*

an approach to predict impacted files from a change request definition has been presented. The approach exploits information retrieval algorithms performed on code entities, such as source files and lines of code, indexed with free text contained in software repositories. We show, in particular that indexing fine grained entities, improves precision, at the cost of indexing a much higher number of code entities.

The empirical validation conducted on three open source projects has given promising results. However, quality of text and project maturity are two factors that strongly impact the performance of the approach. Sometime CVS comments are used for communication rather than for description purpose and in almost all projects there is an initial period of transition that generates noise in both CVS and Bugzilla repositories. Indexes can be build, effectively, only for mature projects for which a huge amount of historical data is available. For young and immature projects this approach fails.

We feel that a direction of improvement should be the introduction of a filter that selects the text to index. The filter should be able to select only the text that well describes the indexed code entities. As a very simple example, CVS comments regarding maintenance and merged revisions should be discarded because, usually, they have not useful information for indexing.

The open source community uses other repository for knowledge sharing, such as: mailing lists, newsgroups, and IRC conversations. They are rich of free text and it should be interesting to investigate how this information can be used in conjunction or as an alternative to CVS and Bugzilla.

8. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301. IEEE Computer Society, 1993.

[3] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*. IEEE Computer Society, 2005.

[4] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. In *ICSM '01: Proceedings of 17th IEEE International Conference on Software Maintenance*, page 364. IEEE Computer Society, 2001.

[5] F. Crestani, M. Lalmas, C. J. V. Rijsbergen, and I. Campbell. Is this document relevant?...probably: a survey of probabilistic models in information retrieval. *ACM Comput. Surv.*, 30(4):528–552, 1998.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of 19th IEEE International Conference on Software Maintenance*, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society.

[7] K. Fogel and M. Bar. *Cross-Validatory Choice and Assessment of Statistical Predictions (with Discussion)*, volume 36. J. the Royal Statistical Soc., 1974.

[8] K. Fogel and M. Bar. *Open Source Development with CVS*. Coriolis, 2001.

[9] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[10] K. S. Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808, 2000.

[11] M. Kamkar. An overview and comparative classification of program slicing techniques. *J. Syst. Softw.*, 31(3):197–214, 1995.

[12] M. Lindvall and K. Sandahl. How well do experienced software developers predict software change? *J. Syst. Softw.*, 43(1):19–27, 1998.

[13] W. Miller and E. W. Myers. A file comparison program. *Software Practice and Experience*, 15(11):1025–1040, 1985.

[14] M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *IEEE 27th International Conference on Software Engineering - The 2nd International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[15] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice-Hall, Upper Saddle River, NJ, 1998.

[16] M. F. Porter. *An algorithm for suffix stripping*. Morgan Kaufmann Publishers Inc., 1997.

[17] B. Ribeiro-neto and Baeza-yates. *Modern Information Retrieval*. Addison Wesley, 1999.

[18] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Transactions on Software Engineering*, 30:574–586, Sept. 2004.

[19] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *IEEE 27th International Conference on Software Engineering - The 2nd International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[20] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.

[21] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *IEEE 26th International Conference on Software Engineering - The 1st International Workshop on Mining Software Repositories*, pages 2–6, 2004.

Are Refactorings Less Error-prone Than Other Changes? *

Peter Weißgerber
University of Trier
Computer Science Department
54286 Trier, Germany
weissger@uni-trier.de

Stephan Diehl
University of Trier
Computer Science Department
54286 Trier, Germany
diehl@acm.org

ABSTRACT

Refactorings are program transformations which should preserve the program behavior. Consequently, we expect that during phases when there are mostly refactorings in the change history of a system, only few new bugs are introduced. For our case study we analyzed the version histories of several open source systems and reconstructed the refactorings performed. Furthermore, we obtained bug reports from various sources depending on the system. Based on this data we identify phases when the above hypothesis holds and those when it doesn't.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics

General Terms

Algorithms, Management, Measurement

Keywords

Refactoring, software evolution, reverse engineering, and re-engineering

1. INTRODUCTION

Changes to source code can be roughly categorized as bug fixes, feature extensions, and refactorings. Intuitively, we expect that feature extensions are more error-prone than bug fixes or refactorings. By definition, a refactoring should not alter the program behavior at all and, thus, not introduce any new bugs. Thus, during phases in the program development where refactorings prevail, we would expect that less errors are introduced than in other phases. In this paper we present the results of a first case study which relates the percentage of refactorings per day to the ratio of bugs opened within the next 5 days.

The remainder of this paper is organized as follows: In Section 2 we describe how we reconstruct refactorings and other changes from version archives. Then, we explain in Section 3 how we relate these changes to bugs that have emerged in the lifetime of a software system. For the case study in Section 4 we applied our technique to three open-source systems. Section 5 discusses related work and Section 6 concludes this paper.

2. OBTAINING REFACTORINGS

In this section we briefly explain our technique to extract refactorings from software version archives such as CVS. The details of this technique can be found in [5].

A prerequisite for detecting refactorings that occurred during the evolution of a software system is to collect information about all changes that have been stored in the archive. After that, these change can be analyzed to decide if they are a refactoring, part of a refactoring or no refactoring at all.

As we focus in this paper on the question of whether refactorings are less error-prone than other changes, we have to compute the ratio of refactorings to other changes. In particular, this allows us to look for those days (or weeks, months, ...) which had a high refactoring ratio.

2.1 Recovering Basic Change Information

To recover the changes performed to a software system during its evolution we analyze the software archive as presented in [9]. As a result we obtain the following information:

- **Versions**: A version describes one revision of one file in the software archive, for example the revision 1.2 of the file `src/Main.java`. For each version we extract the following information: the filename and the revision number, the revision number of the predecessor version (i.e. the version of the same file that has been changed to create this particular version), the developer who checked-in the version into the archive, the log message and the timestamp of the check-in, the state of the version (e.g., "dead" for versions that are not used anymore), the numbers of added, altered, and deleted lines, and the version text.
- **Transactions:** A transaction is defined as the set of versions that have been checked-in into the version archive by an author in one commit operation. As

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

MSR'06, May 22–23, 2006, Shanghai, China.

CVS splits each commit operation containing multiple versions into separate check-ins for each version, we use a sliding time window heuristic to recover transactions quite precisely. For every transaction recovered this way, we additionally store the timestamp of the start, the length, as well as the committer and the log message of the transaction.

For our study we use the transactions as a heuristics which changes have been performed at the same time and, thus, might be related to each other. For our purpose, we are not interested in the numbers of the lines that have been changed, but in the names of the affected code-blocks — these are classes, fields and methods. Figure 1 gives an overview of the overall process of computing these numbers which is described in the following sections.

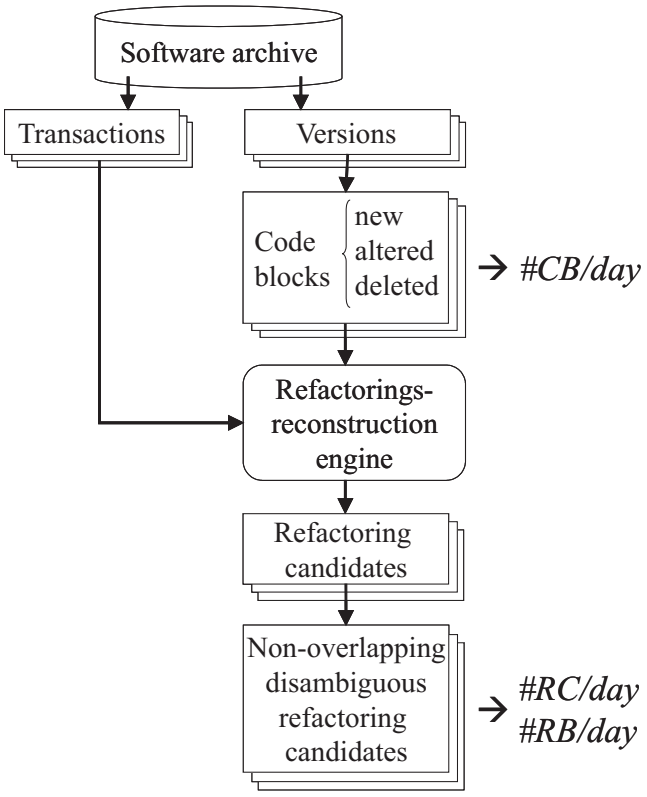


Figure 1: Computation of Changed Blocks and Refactorings

2.2 Finding Changed Code Blocks

To determine which code blocks have been changed in a version with respect to its predecessor, we first have to compute the blocks which are actually contained in it. For this we compute for each version v the following sets:

- the set C_v of classes / interfaces (identified by their fully-qualified class name),
- the set F_v of fields (identified by their signature, consisting of name and type),
- and the set M_v of methods (identified by their signature consisting of name, parameters, and type).

For each block we additionally store the start line and the end line. This allows us to reconstruct the nesting of the symbols in a file, and to distinguish methods that have the same signature but belong to different classes defined in the same file.

Let $B_v = C_v \cup F_v \cup M_v$ the set of blocks contained in version v.

Next, for each version v we compare its set of code blocks with the set of code blocks for the predecessor version v' (for versions that have no predecessor, we take the empty version as v', thus we compare with the empty set). Thus we compute the following sets:

- $B_v - B_{v'}$: new code blocks, i.e. blocks that only exist in the newer version
- $B_v \cap B_{v'}$: common code blocks, i.e. blocks that exist in both versions
- $B_{v'} - B_v$: deleted code blocks, i.e. blocks that only exist in the predecessor version

Obviously, every new code block and every deleted code block is affected by the transition from v' to v. Moreover, common code blocks may also have been altered and, thus, affected by the transition from v' to v.

In our current implementation we use a light-weight regular-expression based parser to compute the code blocks for JAVA files, and perform a textual comparison to find out if a code block has been altered in the transition from version v' to its subsequent version v. Another possibility to get this information is to use the ECLIPSE [8] structural compare plug-in which is able to identify and compare blocks for several file types and programming languages.

2.3 Refactorings

In the previous section we have explained how we extract information about changed JAVA code blocks (classes, methods, field). As we have also recovered transactions we now know which blocks have been affected by changes performed at the same time. Next, we want to determine which of these changes are refactorings, and thus, which of the changed blocks are affected by refactorings.

Our refactoring reconstruction engine [5] takes the information about changed blocks and transactions as input and yields for each transaction a set of refactorings. We call these refactorings *refactorings candidates*, to indicate that they have possibly been performed in this transaction. In this paper, we address refactoring kinds for which candidates can be computed by comparing the signatures and contents of added and removed code blocks. Other refactoring kinds require more semantic information like type inferences or the class hierarchy. For this study we compute refactoring candidates of the following kinds:

- Move/Rename class/interface $c_1 \Longrightarrow c_2$, where c_1 is the old fully-qualified class name and c_2 the new one.
- Move field $(f, c_1) \Longrightarrow (f, c_2)$, where f is the field signature, c_1 is the fully-qualified name of the old class of the field, and c_2 is the fully-qualified name of its new class.
- Move method $(m, c_1) \Longrightarrow (m, c_2)$, where m is the method signature, c_1 is the fully-qualified name of the

old class of the method, and c_2 is the fully-qualified name of its new class.

- Rename method $m_1 \Longrightarrow m_2$, where m_1 is the old signature of the method and m_2 the new one.
- Hide/Unhide method m, where m is the signature of the method that has been hidden respectively unhidden.
- Add/Remove parameter to/from method $m_1 \Rightarrow m_2$, where m_1 is the old method signature and m_2 the new method signature.

However, in this paper we are mainly interested in how many refactorings are performed, but not in the kind of each refactoring. To count the number of refactorings the following issues must be considered:

Overlapping refactorings Our refactoring reconstruction engine both detects refactoring candidates on class level (move/rename class/interface) as well as on a fine-grained level (refactorings on fields and methods). The problem with this is, that refactorings on class level automatically *include* refactorings on the fine-grained level: If a class is moved or renamed, automatically all fields and methods in it are moved to the new location and, thus, are counted. So if we counted both refactorings candidates on class level and on the fine-grained level, we would count some refactorings twice. To solve this problem, we omit the class-level refactoring candidates in our count.

Ambiguous refactorings As the refactoring reconstruction engine only yields refactoring *candidates* (because it cannot decide whether a program transformation really preserves the program behavior or not), for the same transaction several refactoring candidates may be suggested that are ambiguous and may exclude each other. For example, assume that the reconstruction would suggest the following refactoring candidates:

1. Move Method $(m, c_1) \Rightarrow (m, c_3)$.
2. Move Method $(m, c_1) \Rightarrow (m, c_4)$.
3. Move Method $(m, c_1) \Rightarrow (m, c_5)$.
4. Move Method $(m, c_2) \Rightarrow (m, c_3)$.

In this example the first and fourth candidate are ambiguous concerning the source. If both are taken into account, this would mean that two different methods (the one with signature m in class c_1 and the one with the same signature in c_3) are moved to the same target. Although such operations are possible (the methods may have been identical) it is likely that at most one of both candidates is correct. A similar problem occurs with the first three candidates: There are three alternatives to which class the method has been actually moved.

Thus, instead of counting the number of all refactoring candidates, we compute the number of unambiguous refactoring candidates. A conservative approximation for this number is to take as many refactoring candidates into account as there are different sources respectively targets, depending on which number is smaller. In the example above there are two different sources, namely (m, c_1) and (m, c_2), and three different targets: (m, c_3), (m, c_4), and (m, c_5). Thus the number of unambiguous refactoring candidates for this example is 2.

Number of affected blocks The number of affected blocks of a refactoring only depends on the kind of refactoring: Refactorings of kinds move field, move method, rename method, add parameter, and remove parameter affect two blocks because they change the signature of the refactored artifact and, thus, create a new block. In contrast, refactorings of kind hide/unhide method do not change the signature of the refactored block and, therefor, affect only one block.

3. RELATING REFACTORINGS TO BUGS

As the goal of this study is to see whether the common belief, that refactorings are less risky than other changes, is really true, we try to estimate how many issues have emerged because of a change. To this end, we first describe how we get information about the bugs that appeared in a software system during its lifetime. Then we describe how we relate changes and refactorings to these bugs.

3.1 Obtaining Bug Information

Depending on the project, information can be obtained from the following sources:

Bug databases: The most obvious way to gather bug information is to directly access the bug database. Unfortunately, only project administrators have direct access to the database, the average developer and other people have to use a web interface to query and alter the bug database. Querying the database for every existing issue over the web interface can be tedious if we want to retrieve all information available for all bugs. However, getting an overview on the bugs is easy using the web interface. This overview contains for every bug at least its ID, and a hyperlink to additional information about the bug, that can be downloaded and parsed if needed. For SOURCEFORGE projects the overview also includes the summary of each bug, the date when the bug has been opened, the priority, the status of the bug, the developer it is assigned to, and the developer who has submitted it. In contrast, the BUGZILLA overview does not contain the bug open date and the submitter, but instead the severity of the bug, the affected computer platforms, and for closed bugs also the resolution (which may be "bugs is fixed" or "bug report was invalid").

Bug report mails: Bug report mails are emails that are automatically generated and sent to the developer mailing list by bug databases such as BUGZILLA or SOURCEFORGE when a new bug has been opened in the database or the entry of an existing bug has been changed. A bug report mail at least contains information about the ID of the bug that has been altered, and a textual description what exactly has been changed (e.g., the bug has been resolved, or a comment has been added).

To get bug information using bug mails, one needs to have access to the email archive of the project. Fortunately, for most projects—especially those hosted at

SOURCEFORGE—these archives are freely accessible over the web.

Recognizing the bug report mails in the mail archives is quite simple: the sender of this mail is the bug database and normally, the subject has a special format. For example, the SOURCEFORGE bug database sends all its mails with the sender `noreply@sourceforge.net` and the subject "`[ project-Bugs-ID ] bug description`".

Next, we parse the bug report mails using a regular-expression based parser. In this work we are mainly interested in when a bug has been opened. But other information can also be recovered using appropriate regular expressions.

3.2 Relating Changes and Bugs

Unfortunately, although bug information can be retrieved from bug databases and bug report mails as described above, there are only very few cases where the bug information contains exact specifications about which source code change is responsible for the respective bug. Thus, we have to use heuristics here as well.

Our heuristics is to assume that a bug may be caused by changes that have been done in a time window of n days before the bug has been opened (reported).

Thus, for every day d in the lifetime of a project, we compute the number of bugs $\#OB_d^n$ that have been opened at day d and the next n days.

Furthermore, we have to take into account that there are days when there is more activity and when there is less. As a measure for the activity we use the number of blocks changed per day.

Thus, for every day d the number of changed blocks is $\#CB_d = \sum_{t \in T_d} \#CB_t$ where T_d is the set of transactions on day d.

Instead of looking at the absolute numbers of refactorings and bugs per day, we relate them to the number of changed blocks. We also relate $\#OB_d^n$ to the number of changed blocks to be able to compare it with the refactoring ratio.

As the refactoring ratio is a percentage, its values are in the range of [0, 1]. To be able to draw a single diagram containing the refactoring ratio, as well as the number of changed blocks and the number of bugs per block, we normalize the latter two by dividing by the maximum value.

Thus, in our study we compared the following three metrics:

Normalized number of changed blocks:
$\%CB_d = \frac{\#CB_d}{\#CB_{max}}$ where
$\#CB_{max} = max\{\#CB_d | d \text{ day in the projects lifetime}\}$.

Normalized number of bugs per changed block:
$\%BB_d^n = \frac{\#OB_d^n}{\#CB_d \#OB_{max}^n}$ where
$\#OB_{max}^n = max\{\#OB_d^n | d \text{ day in the projects lifetime}\}$.

Number of refactorings per changed block:
$\%RB_d = \frac{\#RC_d}{\#CB_d}$ where $\#RC_d$ is the number of non-overlapping, disambiguated refactoring candidates for day d.

Project	in CVS since	#txns	#versions	#dev
ARGOUML	1998-01-26	16138	65593	42
JEDIT	2001-09-02	2141	10726	6
JUNIT	2000-12-03	832	1707	6

Figure 2: CVS data for the analyzed project txns = transactions, dev = committers

4. CASE STUDY

4.1 The Analyzed Projects

In our case study we have applied the described techniques to three different open source projects: JEDIT, JUNIT, and ARGOUML. Although these projects may not be representative for all open-source projects, they are very different in age, size, number of transactions, and number of involved committers, as Figure 2 illustrates.

4.2 Study Settings

We applied our technique described in the previous sections to these three projects as follows: First, we computed for each transaction the number of total changed code blocks and the number of blocks affected by refactorings. To get information on the level of days, we computed the sum of the number of changed code blocks as well as of the number of code blocks affected by refactoring for each transaction that has been started at the same day.

Next, we collected data about how many bugs have been opened per day for these projects. For ARGOUML a developer created bug statistics [7] and kindly provided the raw data of these to us. For JEDIT we relied on the bug report mails that are sent by the project's bug database to the developer mailing list. As JUNIT does not use such bug report mails we extracted the bug information for this project from the SOURCEFORGE web interface.

As explained, we computed the value $\%BB_d^n$ that relates the changes of day d to bugs opened within the next n days. Obviously, looking only at the same day would not be sufficient. Thus, we decided to use a longer time window of $n = 5$ which roughly corresponds to a working week.

4.3 ARGOUML

For ARGOUML Figure 3 shows for each day the normalized number of changed blocks $\%CB$, the percentage of refactored blocks per day $\%RB$, and the normalized number of bugs per changed block $\%BB$.

It can easily been seen that for most days the percentage of refactorings with respect to all changed blocks is rather small, interestingly there is no day where all changed blocks are affected by refactorings. The days with the highest refactoring percentages are mainly between April 2005 and October 2005, thus we look at this period in more detail (see Figure 5).

When we look at this figure it seems that for most days with a high refactoring rate, the value of $\%BB$ does not change tremendously. But, interestingly, for June 30, the day with the overall highest refactoring rate (98.4%), it increases noticeably. The log messages for the two transactions performed on that day state that the undo functionality is moved to a package on its own, but that a new word-wrap feature is introduced additionally.

Another day with a noticeable refactoring rate and increasing $\%BB$ is July 31. On the transactions at this day,

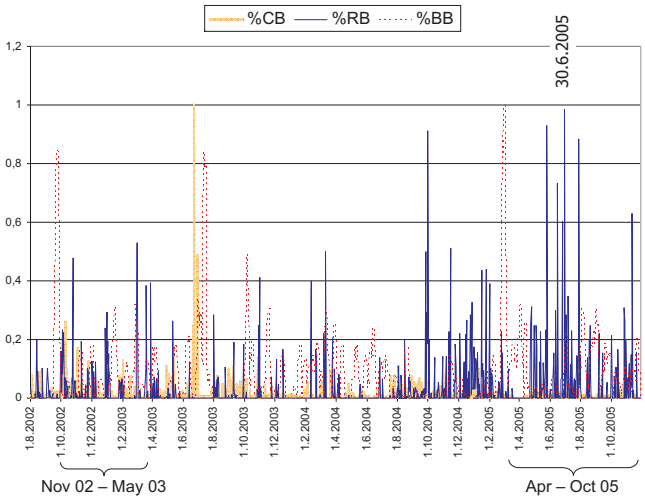


Figure 3: Overview of values computed for ARGOUML

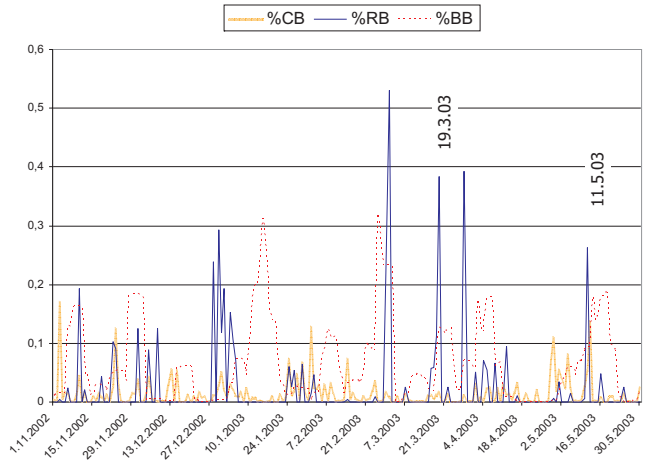


Figure 4: ARGOUML: November 2002 to May 2003

according to the log messages, the class `PropPanelSignal` has been refactored, and furthermore among other things, the developers have "implement(ed) signals and timexpressions for events". When looking at the bugs that have been filed within the next five days we found a bug with the summary "Attributes disappear after typing an initial value" in the component `PropertyPanel`, which also contains the refactored class.

Another phase that contains many refactorings (although the ratio is not very high for these days) is between November 2002 and May 2003—this phase is shown in Figure 4. For most days with a high refactoring rate we cannot see an effect on the normalized number of bugs per changed block, with two exceptions: On March 19 and on May 11 we detected a high refactoring rate (39% resp. 26% of the changed blocks affected by refactorings) but the value of $\%BB$ increases as well. We looked in detail at these two dates: the most noticeable specific characteristic of these days is that they contain quite many transactions: On average between 5 and 6 transactions have been performed per day, but on these days the transaction count has been 16 respectively 21.

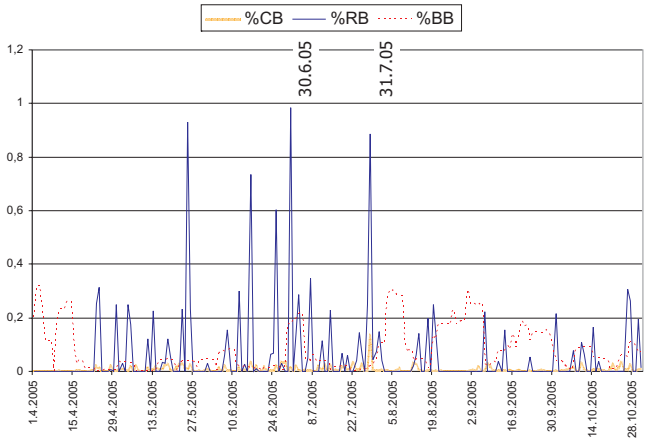


Figure 5: ARGOUML: April to October 2003

4.4 JEDIT

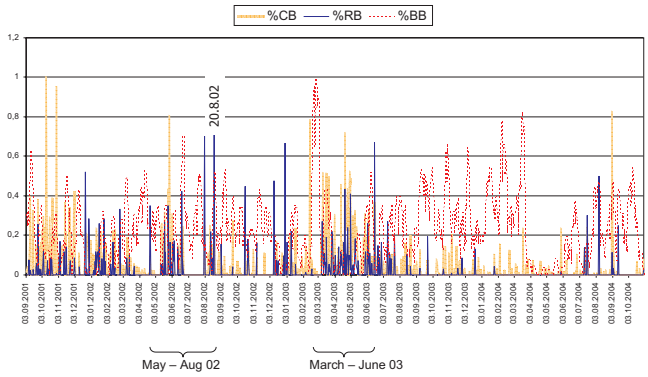


Figure 6: Overview of values computed for JEDIT

Figure 6 illustrates our results for JEDIT. Again it catches our eyes that there is no day that contains only refactorings covered by the kinds we recognize. The highest refactoring rate is even lower than $\frac{3}{4}$: it is 73.4% at August 20 2002.

We zoomed into two periods that seemed to be interesting because there are many refactorings: Figure 7 gives a closer look at the period between May and August 2002, while Figure 8 shows the period between March and June 2003. The following paragraphs describe these two phases in more detail.

4.4.1 *JEDIT refactoring phase in 2003*

Let us first look at the refactoring phase in 2003 (Figure 8): It attracts attention that between April 19 and May 3 a lot of changes with a high percentage of refactorings have been done, but no, respectively very few, new bugs have been introduced in these days. This seems to support the thesis that refactorings are changes that are not error-prone. We inspected the log messages given for the concerned transactions and found out that the developers documented mainly fixes and refactorings, but only few new features in this phase.

There are also other days between March and June 2003 when changes with a high percentage of refactorings have been done and no—or even a decreasing—effect on the value

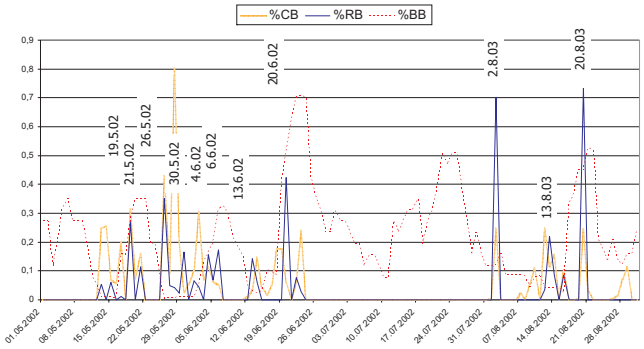


Figure 7: JEDIT: May to August 2002

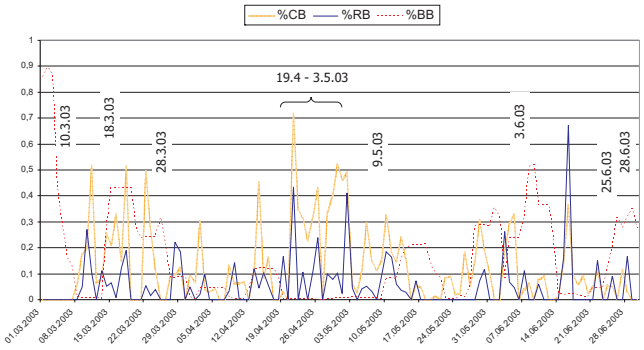


Figure 8: JEDIT: March to June 2003

of $\%BB$ can be observed: The respective days (March 3, March 18, March 28) are annotated in Figure 8.

But we also found days where the refactoring percentage $\%RB$ is rather high but the normalized number of bugs per changed block increases nevertheless. This holds, for example, for May 9, June 7 until June 9, and June 25 until June 28. We manually looked at the log messages of the transactions performed on these days, as well as on the source code changes. Although the refactoring percentage is high for these days, functionality changes have been performed, even within the blocks affected by refactorings.

4.4.2 *JEDIT refactoring phase in 2002*

Figure 7 shows the refactoring phase from May 2002 until the end of August 2002. There are some days (May 21, May 26, May 30, June 13, Aug 8, Aug 13) with a high refactoring ratio that, as expected, do not cause the normalized number of bugs per changed block to increase noticeably.

But for four days (May 19, Apr 6, May 20, Aug 20) with a refactoring percentage of greater than 10%, the value of $\%BB$ increases. At the first of these dates, more than a quarter of the changed blocks have been affected by refactorings. Two days later again 10% of the changed blocks have been affected by refactorings. Nevertheless, the normalized number of bugs per changed block has a peak at these days. We looked in the log messages of the respective transactions to find evidence for the author's intention of the changes and found that there has been a "display code rewrite", "syntax and text area reworking" and several "TokenMarker code refactorings" and "syntax refactoring" (all transactions have been performed by the same developer). Interestingly, one of the bugs that has been filed within 5 days has the summary "text area and syntax packages: Major redraw issues". However, the developer who committed the changes was expecting such problems: He has commented the bug: "The code in CVS is a work in progress. It might not even compile [...] it might not even run. [...] I'm [...] rewriting [...] the syntax highlighting code; so problems [...] are to be expected."

4.5 JUNIT

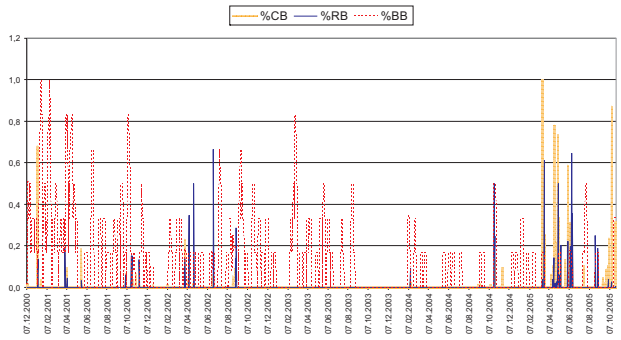


Figure 9: Overview of values computed for JUNIT

We repeated our experiments for JUNIT, the complete results are illustrated in Figure 9. Like for the other projects, there is no day with a refactoring percentage of 100%. The top refactoring percentage for JUNIT is even smaller than for the other two analyzed projects: on June 25 2002 exactly $\frac{2}{3}$ of all changed blocks have been affected by refactorings.

Although we found only refactorings at a few days, there seem to be two phases when refactorings have taken place: The first one between March and September 2002, and the second one between March and October 2005.

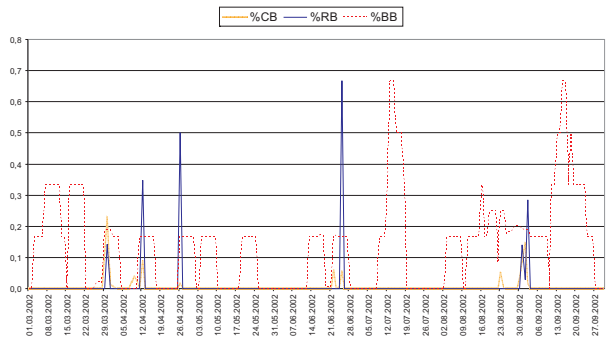


Figure 10: JUNIT: March to September 2002

Figures 10 and 11 show these two phases in more detail. It seems that in 2002 days with a high refactoring rate are likely to be followed by new bug reports, while in 2005 refactorings are rather done after bug reports have been filed.

5. RELATED WORK

Several techniques for detecting refactorings that occurred between subsequent versions of a software system have been developed [2, 4, 1, 5].

In a previous paper we showed that extracted refactoring candidates can be checked for completeness, i.e., whether all

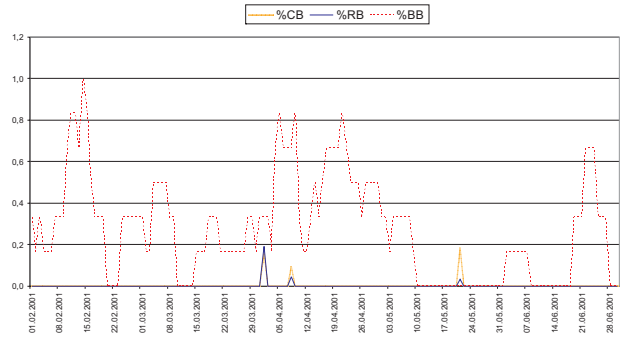


Figure 11: JUNIT: March to October 2005

related locations have been changed [6].

In a recent case study Dig and Johnson found that 80% of API changes, that lead to errors in the applications using these APIs, are refactorings [3].

6. CONCLUSIONS

In this paper we have presented a technique to relate refactoring candidates that we extracted from CVS archives for JAVA programs to bug data in order to find out if the ratio of refactorings with respect to all changes has an impact on the number of bugs that arise in the next days. Although we found interesting correlations between refactorings and bug reports, we are aware that these could be accidental or caused by other factors like feature freezes that we did not yet take into account.

In our case study we applied this technique to three open-source projects. It turned out that in all three projects, there are no days which only contain refactorings. This is quite surprising, as we would expect that at least in small projects like JUNIT there are phases in a project when only refactorings have been done to enhance the program structure. But actually by far the highest refactoring ratio occurred in ARGOUML which is by far the largest one of the projects.

Finally, we found phases of the projects where a high ratio of refactorings was followed by an increasing ratio of bugs, as well as phases where there was no increase. While phases of the second type prevail, phases of the first kind give interesting insight when and why refactorings can cause errors.

7. ACKNOWLEDGMENTS

Michael Stockman kindly provided the bug data for ARGOUML.

8. REFERENCES

[1] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of 7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6-7 September, Kyoto, Japan*, pages 31–40. IEEE Computer Society, 2004.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, pages 166–177, Minneapolis, Minnesota, USA, 2000. ACM Press.

[3] D. Dig and R. Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 389–398, Budapest, Hungary, 2005. IEEE Computer Society.

[4] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[5] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of International Workshop on Program Comprehension (IWPC05)*, St. Louis, Missouri, USA, May 2005.

[6] C. Görg and P. Weißgerber. Error Detection by Refactoring Reconstruction. In *Proceedings of International Workshop on Mining Software Repositories MSR 2005*, St. Louis, Missouri, USA, May 2005.

[7] M. Stockman. ARGOUML statistics and diagrams homepage. `http://user.tninet.se/~zaa397e/argouml/`.

[8] The Eclipse Foundation. Eclipse Homepage. `http://www.eclipse.org`.

[9] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR04)*, Edinburgh, Scotland, UK, May 2004.

Predicting Defect Densities in Source Code Files with Decision Tree Learners

Patrick Knab, Martin Pinzger, Abraham Bernstein
Department of Informatics
University of Zurich, Switzerland
{knab,pinzger,bernstein}@ifi.unizh.ch

ABSTRACT

With the advent of open source software repositories the data available for defect prediction in source files increased tremendously. Although traditional statistics turned out to derive reasonable results the sheer amount of data and the problem context of defect prediction demand sophisticated analysis such as provided by current data mining and machine learning techniques.

In this work we focus on defect density prediction and present an approach that applies a decision tree learner on evolution data extracted from the Mozilla open source web browser project. The evolution data includes different source code, modification, and defect measures computed from seven recent Mozilla releases. Among the modification measures we also take into account the change coupling, a measure for the number of change-dependencies between source files. The main reason for choosing decision tree learners, instead of for example neural nets, was the goal of finding underlying rules which can be easily interpreted by humans. To find these rules, we set up a number of experiments to test common hypotheses regarding defects in software entities. Our experiments showed, that a simple tree learner can produce good results with various sets of input data.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

Keywords

Data Mining, Defect Prediction, Decision Tree Learner

General Terms

Measurement, Management, Reliability

1. INTRODUCTION

A successful software project manager knows how to direct his resources into the areas with the highest impact on the bottom line. Regarding the quality of a software system, the areas with great impact are the parts of the code base with the highest defect density, or even better, with the most future problem reports. Problem reports obtainable from issue tracking systems (*e.g.,* Bugzilla) can be used to assess the perceived system quality with respect to defect rate and density. The objective of such an assessment is to identify the code parts (*i.e.,* software modules) with the highest defect density. Improving them will allow the software developers to reduce the number of problem reports after delivery of a new system or an update.

Our long term goal is to provide software project teams with tools allowing a manager to invest resources proactively (rather than reactively) to improve software quality before delivery. In this paper we address the issue of predicting defect densities in source code files. We present an approach that applies decision tree learners to source code, modification, and defect measures of seven recent source code releases of Mozilla's content and layout modules. Using this data mining technique we conduct a series of experiments addressing the following hypotheses:

1. *Hyp 1*: We can derive defect-density from source code metrics for one release.
   This hypothese covers two sub hypotheses concerned with code quality assessment.
   - *Hyp 1a*: Large source code files have a higher number of defects than small files.
     This is a popular premiss with the underlying assumption that large files are complex, hard to understand and therefore more susceptible to defects. However, there is little to gain here. Even if we assume a balanced distribution of defects, larger files trivially have more defects. More interesting is the defect-density, *i.e.,* number of problem reports per line of code. Which gives us:
   - *Hyp 1b*: Larger files have a higher defect-density.
2. *Hyp 2*: We can predict future defect-density.
   This is the holy grail of software project management. If we can predict the files which will have the highest defect rate in a future release, this would certainly help with ressource allocation in a project.
3. *Hyp 3*: We can identify the factors leading to high defect-density.
   Knowing locations with highest defect density the next step is concerned with gaining insights into the reasons that lead to defects. These insights allow software developers to proactively improve the system and reduce the number of post-release defects.
4. *Hyp 4a*: Change couplings contain information about defect-density in source files of a single release.

MSR'06, May 22–23, 2006, Shanghai, China.

Change coupling has shown to provide valuable information for analyzing change impact and propagation [13, 15]. In this work we take into account the measure of the change coupling strength and test its defect density predictive capability in a single release and:

5. *Hyp 4b*: Change couplings contain predictive information about the number of defects in future releases.

Our experiments showed, that a simple tree learner can produce good results with various sets of input data. We found that common rules of thumb, like lines of code are of little value for predicting defect densities. On the other side, "yesterday's weather" [6], that is, number of bug reports in the past, was one of the best predictors for the future number of bug reports. We also saw, that when we removed various attributes from the input data, the learning algorithm was able to keep its performance, by selecting other, often surprising, attributes.

The remainder of the paper is organized as follows: Related work is presented in Section 2. Section 3 describes the data we used for our experiments. The experiments including a discussion of the results are presented in Section 4. Section 5 draws the conclusions and indicates areas of future work.

2. RELATED WORK

The need for better guidance in software projects to proactively improve software quality led to several related approaches. In this work we concentrate on predicting defect density as well as the number of defects.

A number of approaches concentrated on using code churn measures (*i.e.,* amount of code changes taking place within a software unit over time) for fault and defect density prediction. For instance, Khoshgoftaar *et al.* [9] investigated the identification of fault prone modules in a large software system for telecommunications. Software modules are defined as fault-prone when the debug churn measure (amount of lines of code added or changed for fixing bugs) exceeds a given threshold. They applied discriminant analysis to identify the fault-prone modules based on sixteen static product metrics and the debug churn measure.

Most recently, Nagappan and Ball [12] presented a technique for early prediction of system defect density based on code churn measures. Their main hypothesis is that code that changes many times pre-release will likely have more post-release defects than code that changes less over the same period of time. Addressing this hypothesis the authors showed in an experiment that their relative (normalized) code churn measures are good predictors for defect density while absolute code churn measures are not. In this paper we also address the issue of total and relative metric values but concentrate on different source code metrics of several releases instead of code churn measures solely. Further we apply machine learning techniques for our defect density prediction instead of using statistical regression models.

Munson *et al.* [11] used discriminant analysis and focused on the relationship between program complexity measures and program faults which are found during development. Besides lines of code and related metrics *e.g.,* character count, they use Halstead's program length, Jensen's estimator of program length, McCabe's cyclomatic complexity and Belady's bandwidth metric. Due to the high collinear relationship of these metrics, they mapped them with a principle-components procedure in two distinct, orthogonal complexity domains. They found that, although the detection of modules with high potential for faults worked well, the produced models were of limited value. In our work we use different metrics, especially various coupling metrics (*e.g.,* fan in and fan out). Additionally we build our model from multiple releases with decision tree learners.

Fenton *et al.* [4] tested a range of basic software engineering hypotheses and found that a small number of modules contain most of the faults discovered in pre-release testing and that a very small number of modules contain most of the faults discovered in operation. However, they found, that in neither case it could be explained by the size or complexity of the modules. They distinguished between pre- and post-release fault discoveries, whereas we concentrate on bug reports, which are mostly post-release. We can confirm the findings of Fenton *et al.* regarding the relevance of module size (in our case file size), and their observation concerning the distribution of faults discovered in operation.

In addition to the complexity measures a number of objectoriented software metrics have been developed such as the ones from Chidamber and Kemerer [3]. As with the complexity measures, the results and opinions of the various investigations are different. An early investigation of these metrics comes from Basili *et al.* [1]. They have defined a number of hypotheses regarding the fault-proneness of a class. To validate these hypotheses they conducted a student's project in which the students had to collect data about the faults found in a program. Based on this data they used univariate logistic regression to evaluate the relationship of each of the metrics in isolation and fault-proneness and multivariate logistic regression to evaluate the predictive capability. The results have shown that all but one of these metrics are useful predictors of fault-proneness.

Ostrand *et al.* [2] used a negative binary regression model to predict the location and number of faults in large software systems. The variables for the regression model were selected using the characteristics they identified as being associated with high fault rates. They also found, that a simplified model only based on file size was only marginally less accurate. We can support the finding that lines of code is a good measure for number of faults, from our research. However, this fact is of little help in the management of the development process. To reduce the overall number of faults, we have to reduce the fault density. The focus of our work is more on the understanding of the factors that lead to faults than the actual fault prediction.

Graves *et al.* [7] developed several statistical models to evaluate which characteristics of a module's change history were likely to indicate that it would see large numbers of faults generated as it is continued to be developed. Their best model, a weighted time damp model, predicted fault potential using a sum of contributions from all the changes to the module in its past. Their best generalized linear model used numbers of changes to the module in the past together with a measure of the module's age. They found, that the number of deltas, *i.e.,* the number of changes was a successful predictor of faults, which is also indicated by our experiments. They also found, that change coupling is not a powerful predictor of faults, which our results also support. By using decision trees we use all available measures to build a model including past modification reports, change couplings and various source code metrics.

Hassan and Holt [8] presented heuristics derived from caching mechanisms to find the ten most fault susceptible subsystems which they tested on several big open source projects. Their heuristics are based on the subsystems that were most recently modified, most frequently fixed, and most recently fixed. Although we did not distinguish between repairing modifications and general modifications, most of the information is also contained in our metrics.

Finally, Mohagheghi *et al.* [10] concentrated on the influence of code reuse on defect-density and stability. They found that reused components have lower defect-density than not reused ones. They did not observe any significant relation between the number of

defects, and component size. They neither found a relation between defect-density and component size. Our results support the second finding, but contradict the first.

3. EXPERIMENTAL SETUP

The data for our experiments stems from seven releases of the content and layout modules of the Mozilla open source project. [1] The modules are: DOM, NewLayoutEngine, XPToolkit, NewHTML-StyleSystem, MathML, XML, and XSLT. For more information on these modules we refer the reader to the module owners web-site [2] of the Mozilla project. The selected releases and their release dates are listed in Table 1.

#	Release	Date
1	0.92	June, 2001
2	0.97	December, 2001
3	1.0	June, 2002
4	1.3a	December, 2002
5	1.4	June, 2003
6	1.6	January, 2004
7	1.7	June, 2004

Table 1: Selected Mozilla releases.

In release 1.7 the seven content and layout modules comprise around 1.300 C/C++ source and header files with a total of around 560,000 lines of code. From this set of files we selected 366 out of 504 *.cpp files. We skipped 138 files because they did not show a complete history as is needed for our experiments (*i.e.,* they were added/removed during this time period). We also skipped the header files (817 *.h files) because they are naturally connected with the corresponding implementation files. So, there is nothing to gain with respect to analyzing the change coupling and predicting the defect density of these source files.

For this set of *.cpp source files per release we computed the source code, modification, and defect report metrics as listed in Table 2. For the source code metrics we parsed each source code release using the Imagix-4D C/C++ analysis tool. [3] The modification and defect report metrics were retrieved from the release history database that we extracted from Mozilla's CVS and Bugzilla repositories as has been presented in our previous work with this project [5].

The first three source code metrics listed in Table 2 quantify the size of a *.cpp file according the lines of code (linesOfCode), the number of defined global and local variables (nrVars), and the number of implemented functions/methods (nrFuncs). The following four source code metrics quantify the strength of the static coupling of a *.cpp file with other *.cpp files. For our experiments we consider incoming (incomingCallRels) and outgoing function calls (outgoingCallRels) as well as incoming (incomingVarAccessRels) and outgoing variable accesses (outgoingVarAccessRels).

The remaining metrics are retrieved from the release history database and computed for the time from the begin of the Mozilla project to the selected release dates. They denote the number of checkins of a *.cpp file (nrMRs), the number of times a file was checked in together with other files (sharedMRs), and the number of reported problems (nrPRs). For the latter metric we further detail the measures into additional categories denoting the different severity levels of reported problems. These levels range from problem reports that are marked as trivial to system critical problem reports (*i.e.,*, system crashes, loss of data). They allow us a more detailed classification of the defects in source files.

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

[2] http://www.mozilla.org/owners.html

[3] http://www.imagix.com

Name	Description
linesOfCode	Lines of code
nrVars	Number of variables
nrFuncs	Number of functions
incomingCallRels	Number of incoming calls
outgoingCallRels	Number of outgoing calls
incomingVarAccessRels	Number of incoming variable accesses
outgoingVarAccessRels	Number of outgoing variable accesses
nrMRs	Number of modification reports
sharedMRs	Number of shared modification reports
nrPRs	Number of problem reports
nrPRsNormal	nrPRs with severity = normal
nrPRsTrivial	nrPRs with severity = trivial
nrPRsMinor	nrPRs with severity = minor
nrPRsMajor	nrPRs with severity = major
nrPRsCritical	nrPRs with severity = critical
nrPRsBlocker	nrPRs with severity = blocker

Table 2: Base metrics computed for a C/C++ file.

The shared modification reports metric (sharedMRs) represents the number of times a file has been checked into the CVS repository together with other files. The reason for adding this metric is that the defect density of a file is higher when modifications (*e.g.,*, bug fixes) are spread over several files instead of being local to one source file. This metric has been used several times in recent investigations to assess the quality of software systems and their evolution (see for example [13, 15]). In this paper we test its defect density predictive capability (see Hyp 4a and Hyp 4b).

The metrics listed above are all computed for each selected release. For predicting the defect density of files we further added trend and normalized values of these metrics. Trends are denoted by the deltas of metric values between two subsequent releases. For instance, the number of functions added/removed or the number of critical problem reports reported from one release to the next. Total as well as delta values are normalized with the size of a file expressed in lines of code (linesOfCode). Such a normalization is a key factor for predicting the defect density namely the number of new defects per line of code.

Total and delta values as well as their normalized values form the input to the experiments presented in the following sections. Regarding the metric names used in the experiments we prefix each metric name with the kind of value: total metrics with "static_"; normalized metrics with "norm_"; and trend metrics with "delta_". Furthermore, the number indicating the release (see Table 1) is added to each metric name. For instance, delta_nrMRs_4 denotes the number of modification reports added from release 1.0 to release 1.3a.

4. EXPERIMENTS

Before we go into our data mining experiments we conducted a number of descriptive statistics analysis with the selected Mozilla releases. Here we present an excerpt of the results we obtained for the Mozilla release 1.0. Similar observations apply to the other Mozilla releases. Concerning Hyp 1a and Hyp 1b the scatter plot in Figure 1 shows that number of problem reports in release 1.0 display a strong linear correlation with lines of code. So big files do not have a higher problem reports to lines of code ratio which shows us that at least for Mozilla the popular belief that big files are

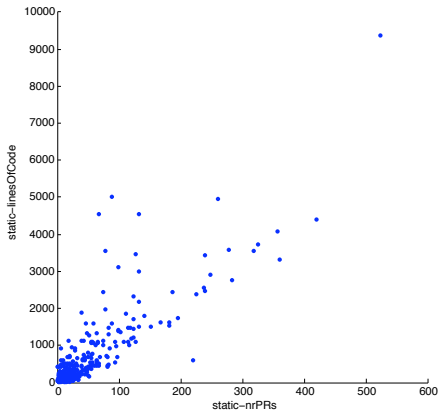


Figure 1: lines of code vs number of problem reports

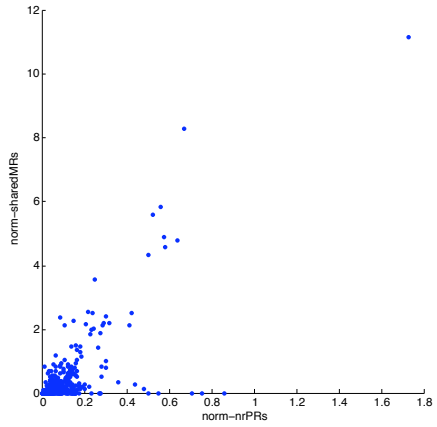


Figure 2: normalized shared modification reports versus normalized problem reports

trouble files does not hold. The downside of this observation is that lines of code does not make for a simple indicator to detect problem files.

Our use of lines of code per file instead of, for example, lines of code per class, results from the fact that most of our metrics like shared modification reports or problem reports, are only calculated for files. Lines of code per class might be of some significance when assessing defect-density, however, this is, based on the research of Fenton and Ohlsson [4], not the case. In the data mining experiments we will further elaborate on the issue whether lines of code has any predictive value.

To test Hyp 4a we analyze the correlation between the normalized values of shared modification reports and problem reports. The scatter plot for Mozilla release 1.0 is shown in Figure 2. The correlation coefficient is 0.7234, which, in combination with the graphic, shows a strong linear correlation between the two values. However, what value, the normalized shared modification reports metric presents for the prediction of future number of defects, remains to be seen.

The process for all data mining experiments is as follows: We export the selected data to an arff file (*i.e.,* a text based data file readable by the WEKA [14] explorer), which is then loaded into the WEKA explorer. We then run the five bins equal frequency discretizer over our data to get the input for our classifier. The use of equal frequency distribution in the discretizer means that the prior probability for an instance falling into a given class is twenty percent. The classifier is the J48 tree learning algorithm provided by the WEKA tool. The accuracy is calculated with ten-fold cross validation.

Exp 1: Problem reports from non PR metrics of the same release: In the first experiment we use all available data from release four (1.3a) excluding problem report metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.) to predict the number of problem reports of release four (nrPRs_4). Figure 3 depicts the top levels of the generated decision tree. We can see, that the attribute with the most information concerning the number of problem reports is the number of modification reports, hence it appears at the root. Attributes on the second level are: added number of modification reports since release 3, shared modification reports, and lines of code. We got

```
Correctly Classified Instances   227   (62.0219 %)
Incorrectly Classified Instances 139   (37.9781 %)
```

which is good, given the prior probability of 0.2. Looking at the confusion matrix

```
 a  b  c  d  e   <-- classified as
 60  9  3  1  0 |  a = '(-inf-7.5]'
 12 40 22  2  0 |  b = '(7.5-15.5]'
  6 22 25 18  0 |  c = '(15.5-25.5]'
  1  3 16 42 11 |  d = '(25.5-62.5]'
  0  0  0 13 60 |  e = '(62.5-inf)'
```

we see the detailed performance for our five classes selected by the discretizer. The top row of the confusion matrix shows the labels of the predicted classes. On the right are the labels and the corresponding intervals of the actual classes. Each cell of the matrix denotes the number of instances (source files) classified as a, b, c, d, or e. The matrix diagonal contains the exact matches. For instance, the numbers in the bottom row state that 60 instances which are of the actual class e (*i.e.,* source files with more than 62.5 problem reports), where classified correctly. 13 instances where wrongly classified as d, none as c, b, or a.

Taking into account only the worst twenty percent, the algorithm gets 82 percent right, and the other 18 percent are put into the second worst class. From a management's point of view, this presents a valuable result. If the manager concentrates his ressources on the files which were classified as e or d (*i.e.,* the worst and second worst class), he would have covered 100 percent of the worst files (*i.e.,* the files with the highest number of defects).

The connection between the number of modification reports and problem reports is not that surprising. If there are many bugs, one has to fix them, which generates modification reports. So what happens if we take the modification reports away from our learning algorithm?

Exp 2: Problem reports from non PR metrics of the same release without MR data: We do the same experiment as above using the available metrics from release four (1.3a) excluding modification report data (*i.e.,* nrMRs, sharedMRs) and problem report metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.). With this data we predict number of problem reports per line of code (norm_nrPRs_4) for release four (1.3a). Normalized problem reports are better suited to assess the badness of a file, because big files with a low defect-density are rated better than small files stuffed with bugs.

Results are below:

```
Correctly Classified Instances      138   (37.7049%)
Incorrectly Classified Instances    228   (62.2951%)
```

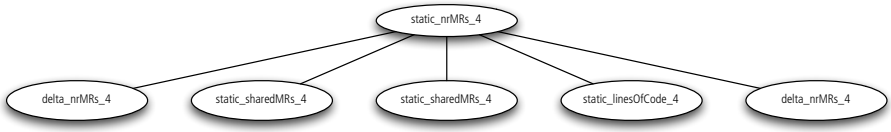


Figure 3: Top levels of decision tree resulting from Exp 1

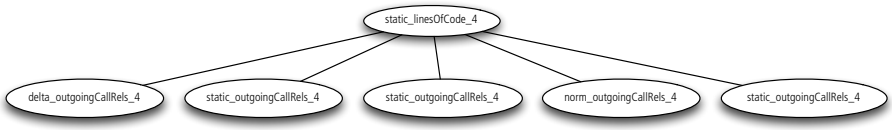


Figure 4: Top levels of decision tree resulting from Exp 2

```
 a  b  c  d  e   <-- classified as
26 20 11  7  9 |  a = '(-inf-0.037225]'
19 20 19  4 11 |  b = '(0.037225-0.065399]'
 9 22 22 10 10 |  c = '(0.065399-0.099327]'
 9 12  9 34 10 |  d = '(0.099327-0.143163]'
 5 15  9  8 36 |  e = '(0.143163-inf)'
```

The quality, although significantly above the prior probabilities, is much worse. Still, by looking at the tree in Figure 4, we can see that there is at least some information in the size of a file for the prediction of the number of problem reports. Although, by looking at the confusion matrix, we can see that the predictions are heavily scattered which makes them hardly useful. For assessing the importance of modification reports data we have to conduct additional experiments.

Exp 3: Normalized problem reports from non PR metrics of the same release: Here we derive the normalized number of problem reports from all, but problem report metrics. Thus repeating experiment one with normalized problem reports as the target class. To predict number of problem reports per line of code (norm_nrPRs_4) of release four (1.3a) we use all metrics from release four except PR metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.). We get:

```
Correctly Classified Instances    192  (52.459 %)
Incorrectly Classified Instances  174  (47.541 %)
```

for this experiment, which confirms the results of experiment one, but differs in at least one important way. Lines of code is not present in the top levels of the resulting tree as shown in Figure 5. In the full tree lines of code is only used in one branch on the third level. This confirms that lines of code is of marginal importance for the prediction of defect-density and lets us reject [4] Hyp 1a, and Hyp 1b.

The confusion matrix illustrates the good performance of the classifier. By looking at the diagonal we see moderate dispersion of the values. If we count near misses, the prediction, especially for class e, is excellent.

```
a  b  c  d  e   <-- classified as
48 19  4  2  0 |  a = '(-inf-0.037225]'
14 33 15  7  4 |  b = '(0.037225-0.065399]'
 6 16 33 13  5 |  c = '(0.065399-0.099327]'
 3  8  7 35 21 |  d = '(0.099327-0.143163]'
 1  2  7 20 43 |  e = '(0.143163-inf)'
```

[4]This is an informal rejection as we have not used any formal hypotheses testing model such as T-test.

Exp 4: Normalized problem reports from non PR metrics of the same release without sharedMR data: Here we exclude shared modification report metrics thus using all metrics except PR metrics (*e.g.,* nrPRs_4, nrPRsMajor_4, etc.) and shared modification report metrics (*e.g.,* sharedMRs, norm_sharedMRs) from release four, to predict the number of problem reports per line of code (norm_nrPRs_4) of release four (1.3a).

```
Correctly Classified Instances    197  (53.8251 %)
Incorrectly Classified Instances  169  (46.1749 %)
```

```
 a  b  c  d  e   <-- classified as
49 17  3  4  0 |  a = '(-inf-0.037225]'
14 35 14  6  4 |  b = '(0.037225-0.065399]'
 3 16 37 13  4 |  c = '(0.065399-0.099327]'
 2  7 13 29 23 |  d = '(0.099327-0.143163]'
 1  5  2 18 47 |  e = '(0.143163-inf)'
```

The error rate and the confusion matrix are almost identical to experiment three. This is a strong sign, that the number of problem reports does not depend on the amount of logical coupling a file has with its surrounding.

But, taking a closer look at Figure 6 we can see that other coupling metrics were used in the prediction of number of problem reports: added normalized outgoing call relationships and incoming call relationships.

The results of these first experiments show, that we can predict defect densities (measured by number of problem reports) with accuracies of more than 50% given a prior probability of 20%. So we can accept Hyp 1. However, lines of code is not a good predictor of defect-density so we have to reject Hyp 1a and Hyp 1b. At this point, we cannot verify Hyp 4a fully. The classifier uses mainly other modification report metrics for the prediction which indicates a low importance of shared modification reports for defect prediction.

The next set of experiments are mainly concerned with Hyp 2 and Hyp 4b.

Exp 5: Added problem reports of release 6 with data from releases 3, 4, 5: For experiment four we use all available data from releases 3, 4, and 5, *e.g.,* lines of code in release three (linesOfCode_3), added modification reports in release four (delta_nrMRs_4), the number of added problem reports with severity major per lines of code in release five (delta_norm_nrPRsMajor5) and predict the number of added problem reports in release 6 (delta_nrPRs_6).

The performance of the classifier is acceptable:

```
Correctly Classified Instances    186  (51.2397%)
```

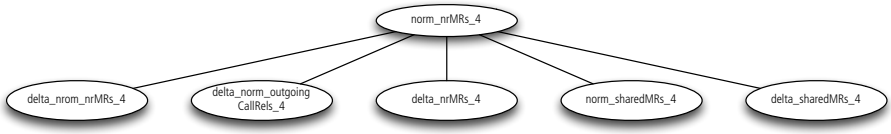


Figure 5: Top levels of decision tree resulting from Exp 3

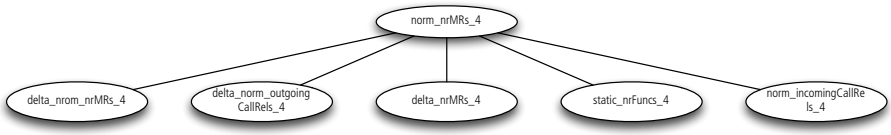


Figure 6: Top levels of decision tree resulting from Exp 4

```
Incorrectly Classified Instances  177  (48.7603%)

 a   b   c   d   e   <-- classified as
 104  11  12   9   2 |   a = '(-inf-0.5]'
  26  16   2   4   0 |   b = '(0.5-1.5]'
  25   4  23  11   4 |   c = '(1.5-3.5]'
  12   3  14  12  15 |   d = '(3.5-6.5]'
   3   0   4  16  31 |   e = '(6.5-inf)'
```

The confusion matrix shows a high accuracy for class a, lower accuracy for the middle classes (b,c,d), and again a high accuracy for class e if we count near misses.

In Figure 7 we see that the top node, added problem reports with severity major from release 4, divides the data set the best regarding added problem reports. The presence of change coupling metrics only in a few lower branches shows that isolated, they are not very valuable for the prediction of the future number of defects. Which supports our finding, that there are no simple dependencies between defect-density and other metrics.

Exp 6: Added problem reports of release 7 with data from releases 3, 4, 5, 6: This experiment is a repetition of Exp 5 but predicting for release seven (delta_nrPRs_7) using input data from releases three through six. As we can see, from the output below, the performance is better and, more interesting, the top node has changed to something, at least for us, surprising. In Figure 8 the top node of the tree is static_nrFuncs_6. At the second level, however, mostly problem report metrics from earlier releases are used.

```
Correctly Classified Instances     215  (59.2287%)
Incorrectly Classified Instances   148  (40.7713%)

  a   b   c   d   e   <-- classified as
 145  17   2   1   2 |   a = '(-inf-0.5]'
  35  14   7   5   5 |   b = '(0.5-1.5]'
   7   8   6   6   2 |   c = '(1.5-2.5]'
   8   7   7  18  12 |   d = '(2.5-4.5]'
   3   4   2   8  32 |   e = '(4.5-inf)'
```

Conducting the same experiment with normalized added problem reports as target attribute, the performance degrades to:

```
Correctly Classified Instances      162  (44.6281%)
Incorrectly Classified Instances    201  (55.3719%)
```

This result supports our assumption that number of functions is used as a measure for the length of the file. When we remove number of functions from the input data of the initial experiment static_outgoingCallRels_6 is at the root of the resulting tree. This suggests that number of functions is somehow related to outgoing calls. However, such a conclusion is premature considering the displayed complex dependencies between the various metrics.

In experiment four and five we showed, that it is possible to predict future defect-density with data mining techniques to an extend that is useful for an engineer or the management. We therefore can accept Hyp 2.

However, as we have seen in the other experiments, the relationships between the various metrics are complex. The sheer amount of data makes it impossible to intuitively understand the underlying decisions of a classifier by just looking at a generated tree. This leeds to the partly rejection of Hyp 3.

5. CONCLUSIONS AND FUTURE WORK

Our long term goal is to provide software project teams with tools allowing a manager to invest resources proactively (rather than reactively) to improve software quality before delivery. A key factor of such tools is the capability to predict defect densities in software modules such as source files or classes.

In this paper we specifically investigated the application of data mining on a number of source code, modification, and defect measures to test their applicability for defect prediction. The focus of our work is more on the understanding of the factors that lead to defects than the actual defect prediction. For this we stated a set of hypotheses that we addressed in a series of experiments with data from seven releases of the content and layout modules of the Mozilla open source project.

The data mining experiments showed, that a decision tree learner (J48) can produce reasonable results with various sets of input data. Regarding our hypotheses:

- We were able to *predict defect densities with acceptable accuracies* with metrics from the same release and therefore accepted Hyp 1.
- We found that *lines of code has little predictive power* with regard to defect-density therefore rejected Hyp 1a and Hyp 1b. In general, *size metrics such as number of functions are of little value for predicting defect densities*.
- We were able to *predict defect-density with satisfactory accuracy* by using evolution data (*e.g.,* number of modification reports) therefore accepting Hyp 2.

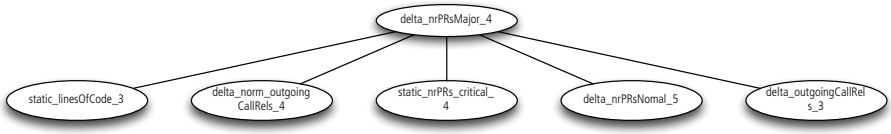


Figure 7: Top levels of decision tree resulting from Exp 5

Figure 8: Top levels of decision tree resulting from Exp 6

- Due to complex relationships between the various metrics we could *only partly identify factors that lead to high defect-density*. This resulted in the partly rejection of Hyp 3.
- We found that *change couplings are of little value for the prediction of defect-density* therefore we rejected Hyp 4a and Hyp 4b.

Future work is concerned with including detailed measures of modifications (*e.g.,* number of statements changed) and defects (*e.g.,* bug status information) in our experiments. In addition, we also plan to take into account the various source code complexity measures, such as McCabe's cyclomatic complexity or the Halstead complexity measures. With this additional information we can gain deeper insights into the internals of the implementation as well as the past defects and modifications that caused increase as decreases of defect densities in source files and classes.

Another area of future work is to use other data mining techniques and conduct our experiments with additional case studies from the open source community as well as industrial software systems.

6. ACKNOWLEDGMENTS

We thank Harald Gall, Peter Vorburger, Beat Fluri and the anonymous reviewers for their valuable input. This work was supported by the Swiss National Science Foundation.

7. REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[2] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[4] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, 2000.

[5] 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, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press.

[6] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, Washington, DC, USA, 2004. IEEE Computer Society.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[8] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[9] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, page 364, Washington, DC, USA, 1996. IEEE Computer Society.

[10] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[11] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Softw. Eng.*, 18(5):423–433, 1992.

[12] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, New York, NY, USA, 2005. ACM Press.

[13] J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal, September 2005. IEEE Computer Society Press.

[14] I. H. Witten and E. Frank. *Data Mining*. Morgan Kaufmann Publishers, 1999.

[15] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.

Information Theoretic Evaluation of Change Prediction Models for Large-Scale Software

Mina Askari
School of Computer Science
University of Waterloo
Waterloo, Canada
maskari@uwaterloo.ca

Ric Holt
School of Computer Science
University of Waterloo
Waterloo, Canada
holt@uwaterloo.ca

ABSTRACT
In this paper, we analyze the data extracted from several open source software repositories. We observe that the change data follows a Zipf distribution. Based on the extracted data, we then develop three probabilistic models to predict which files will have changes or bugs. The first model is Maximum Likelihood Estimation (MLE), which simply counts the number of events, i.e., changes or bugs, that happen to each file and normalizes the counts to compute a probability distribution. The second model is Reflexive Exponential Decay (RED) in which we postulate that the predictive rate of modification in a file is incremented by any modification to that file and decays exponentially. The third model is called RED-Co-Change. With each modification to a given file, the RED-Co-Change model not only increments its predictive rate, but also increments the rate for other files that are related to the given file through previous co-changes. We then present an information-theoretic approach to evaluate the performance of different prediction models. In this approach, the closeness of model distribution to the actual unknown probability distribution of the system is measured using cross entropy. We evaluate our prediction models empirically using the proposed information-theoretic approach for six large open source systems. Based on this evaluation, we observe that of our three prediction models, the RED-Co-Change model predicts the distribution that is closest to the actual distribution for all the studied systems.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement- *Version control*

D.2.8 [**Software Engineering**]: Metrics- *Performance measures, Process metrics*

General Terms
Performance, Reliability, Theory

Keywords
Prediction Models, Evaluation approach, Information Theory

MSR'06, May 22–23, 2006, Shanghai, China.

1. INTRODUCTION
Software systems are continuously being changed to adapt to meet the needs of their users or to correct the faults appearing in systems during development or after deployment. There has been extensive research on new processes and approaches for developing software systems to minimize these new modifications. The idea is that during software development by following some specific principles, the probability of certain kinds of modifications can be decreased. Despite this progress, new changes and bugs are inevitable during software development. However, if software developers were able to forecast the occurrence of changes and bugs then they could mitigate their impact. Therefore, developing accurate techniques to predict the future behavior of changes and bugs can be valuable for software development and maintenance.

The idea for predicting which files/subsystems are most susceptible to having a fault in the near future is a well-known idea. There exist several prediction models [5][6][8][9][10][14] and more are emerging. However, many of these fault prediction models have not been evaluated in practice and some of them are not applicable to large-scale software systems. The majority of fault prediction models are applicable to deployed systems only. The general approach for evaluating these models is to run the system and collect the observed information during its execution and then compare it with the results predicted by the models [18]. The problem is that too often these models are not general and hence, they are not applicable to different software systems. In many cases, because the models measure different metrics, the results are not comparable [18]. There are many questions with respect to the validity of the underlying assumptions, accuracy, and applicability of software prediction models. In this paper, our goal is to contribute toward more general and realistic assessment and prediction of software modifications based on theoretical and empirical studies. We are interested in methods and models that have two properties. First, they use data collected during development process and second, their distance from the actual but unknown distribution of the collected data can be measured. Our goal is twofold: first, to develop prediction models driven by software repositories and second, evaluate and compare different models using a mathematical approach. We use historical records, from source control repositories of large software systems, to develop prediction models and to estimate how much information is captured by the models.

Figure 1 illustrates the problem we are trying to solve. Suppose we have a list of all the events that have so far occurred on different files of a software system during development process.

These events are file changes to fix bugs, or to add new features or change existing features. We have extracted these events from the history of the software. For example f_{21} shows that one modification has happened on file 21 at the specific time (see Figure 1). We ask this question: To what degree are these changes unpredictable? More particularly, what will be the uncertainty of the next sample, if all past samples are known? In some cases, it may be impossible to say anything about the next sample regardless of how many past samples are already known. In other cases, the process may be much less uncertain about the next sample when given the history of the changes. Having extracted historical data, we want to determine how much information this data provides us about the future. Our results indicate that CVS log data contains information about the past that can help to predict the future. The questions include: How much information is buried in the CVS logs and how can we capture this information? How good are the prediction models that use this information to predict the future?

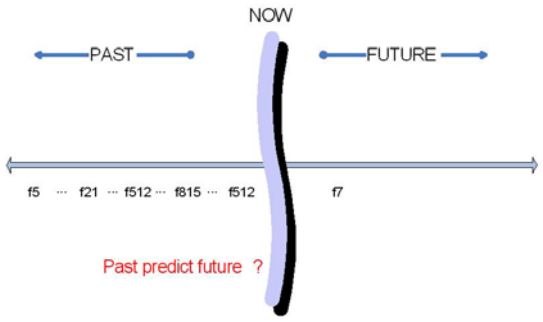


Figure 1. Does past predict future?

After introducing related work in Section 1.1, the rest of the paper is organized as follows. Section 2 presents the techniques and approaches we used to perform different experiments in order to analyze the data extracted from several open source software repositories. In Section 3, we present our three prediction models and describe the steps involved in developing the models. In Section 4, we present an information theoretic approach for evaluation prediction models. In Section 5 presents the results of using two approaches for evaluating our proposed prediction models: the Top Ten List approach proposed by Hassan et al. [8] and our information theory based approach. Finally, Section 6 concludes the paper and discusses possible future works.

1.1 Related Work

Many researchers [2][5][6][8][9][16][17] in software development area have realized the value of historical data and have used them in their research ranging from software design to software understanding, software maintenance, development process and many more areas.

There is considerable research on developing tools to recover such historical data. Hassan et al. [7] developed C-REX tool, an evolutionary code extractor, which recovers information from source control repositories. Zimmermann et al. [19] used version repositories to determine co-change clusters. They applied data mining to version histories in order to guide programmers through related changes. For detecting another kind of co-changes Gall et al. [5] used software repositories. They uncovered the dependencies and interrelations between classes and modules (logical dependencies) which can be used by developers in maintenance phase of a system.

Graves et al. [6] showed that there is a relation between the number of changes a subsystem has with the future faults in that subsystem. Hassan et al. [8] presented various heuristics using historic version control data to create the Top Ten List. Top Ten List highlights to managers the ten most susceptible subsystems to have a fault. They also developed techniques to measure the performance of these heuristics.

Mockus et al. [12] studied a large legacy system to test the hypothesis that historic version control data can be used to determine the purpose of software changes and to understand and predict the state of a software project [13]. Khoshgoftaar et al. [9][10] used process history to predict software reliability and to show that the number of prior modifications to a file is a good predictor of its future faults. Eick et al. [4] presented visualization techniques to explore change data to help engineers understand and manage the software change process. Ostrand, et al. [14] suggested a model to predict the number of faults for a large industrial inventory system based on the history of the previous releases.

Our approach takes guidance from this previous work, but is notably different by suggesting new prediction models and by using an information theoretic approach to measure the effectiveness of such models.

2. CHARACTERISTICS OF THE DATA

The prediction models and the evaluation methods presented in this paper are based on change history data. Change data is the information generated during development process and can be obtained through mining the repositories of the software. We began by analyzing the extracted data to understand its statistical properties. In particular, we observed that history data has Zipf distribution [20].

2.1 Studied Systems

To perform our study we used several CVS logs of open source software systems. Table 1 summarizes the details of the software systems we studied. The oldest system is over ten years old and the youngest system is five years old. We tried to choose the applications from different domains and different sizes. We were looking for any kind of change and bug which happens to different files of a system. The process of acquiring such specific data is very challenging, since CVS logs are mainly designed as record keeping repositories and commits aren't atomic and large amount of data stored in these repositories complicates the data extracting process. For analyzing data and creating prediction models and comparing them based on the data, our main concern was to perform our studies on the data of several CVS logs software systems in a standard format that is easier to process and not developing tools that automatically recover data from these repositories. So we obtained and used the data which were extracted from these CVS logs by tools developed by Hassan et al. [7]. This let us concentrate on analyzing the extracted data instead of spending time developing tools to recover the data.

Table 1. Number of events available for different systems

Application Name	Duration (Month)	Total Events	Bug or Changes	Total Files
OpenBSD	88	80354	67149	7065
FreeBSD	115	126432	101252	5272
KDE	70	93204	77994	4063
Koffice	58	92944	73409	6312
NetBSD	119	239628	131307	11760
Postgres	77	41175	26510	1468

2.2 Zipf's Law

We started by counting the number of modifications which happened for each file during the development process. Based on the history of the development, if we count how often each file is modified, and then list the files in order of the frequency of occurrence, we can explore the relationship between the frequency of a file and its position in the list, known as its rank. Figure 2 illustrates the number of modifications for each file for different systems we studied. Different systems have different number of files. Therefore, to compare all the studied systems in a single plot, we used the percentage of files and percentage of activities for each file in Figures 2 to 4.

As it can be seen from the figure 2, there are few files with high frequency of changes but many files with very low number of changes. It also can be seen in the figure, these frequencies follow a similar pattern in all studied systems. This behavior indicates that the change data follows the general form the Pareto (or 80-20) law [15] and Zipf's law [20].

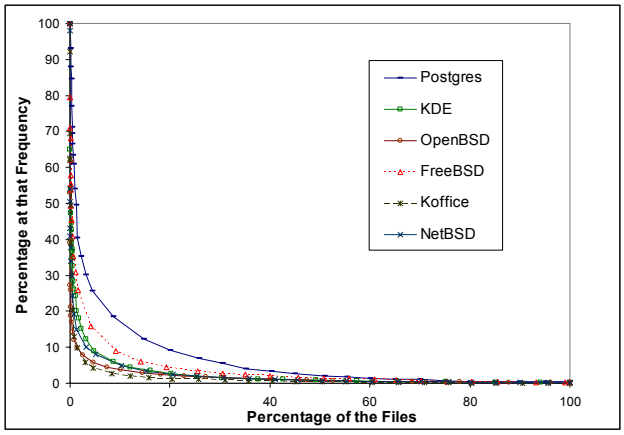


Figure 2. Change data follows Zipf's law.

The Pareto law, in its generalized form, states that 80% of the objectives - or more generally the effects - are achieved with 20% of the means. In order to show that there is 80-20 law in our data, we plotted the cumulative distributions of the file frequencies in Figure 3. It can be seen from the figure that almost 20% of the files in the systems have (almost) 80% of activities during development. To show that Zipf's law holds for the data, we plotted the log-log scale of the cumulative frequency distributions; see Figure 4. It can be seen that the points are close to a single straight line thereby confirming that the data approximates Zipf's law [20].

3. CHANGE PREDICTION MODELS

The prediction of future modifications in a large software system is an important part in software evolution. Since most prediction models in past studies have been constructed and used for individual systems, it has not been practically investigated whether a prediction model based on one system can also predict faults and changes accurately in other systems. Our expectation was that if we could build a model applicable to different range of systems based on the information which is generated during development process, e.g. CVS logs, it would be useful for software developers. In this Section we will show several prediction models which can use the CVS logs to predict the future bugs and changes in any arbitrary system. These models are generally in form of probability models.

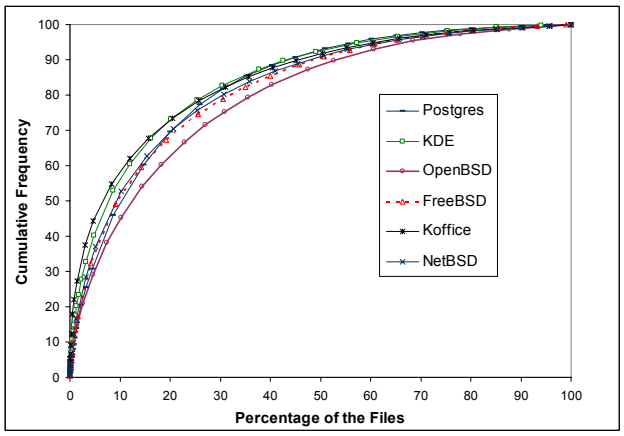


Figure 3. Cumulative frequency distributions.

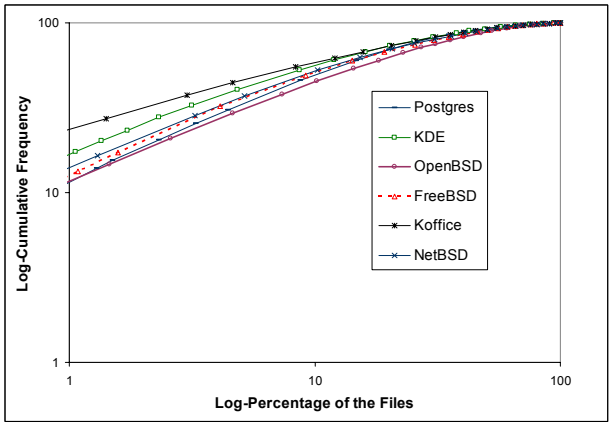


Figure 4. Log-log scale of cumulative distributions.

After extracting the changes and bugs that occurred in the various files of a system during development, we created a sequence of events showing file changes to fix bugs or to add features. Having this sequence of events our goal is to predict future comparable events. There are many files in the systems

we studied, ranging from 1000 to 20000 files. We wanted to construct a probabilistic model of this process, in other words, to define a probabilistic model that characterizes the result of the next element in the sequence. We assume that we know that the possible value space, i.e., the Domain D (i.e., sample space) for event *e* (considered as a random variable). In our work, D is the set of files in the system. We denote the elements of this Domain as $f_1, f_2 \ldots f_m$. Our goal is to define a good probability model to give the probability that the i^{th} (i.e., next) element in the sequence will have a particular value (will be a particular file); in other words for finding the probability distribution of random variable e_i, what we need to do is to decide on the form of the underlying model of the sequence of events. Ideally this would be a conditional probability function of form $P(e_i \mid e_1, e_2, \ldots, e_{i-1})$. Our work is complicated by the fact that, in general, a new probability function is needed for each e_i. Based on this approach, we will now present three probabilistic models.

3.1 Most Likely Estimation (MLE) Model

Our first model, maximum likelihood estimate (MLE), simply uses the counts from the sequence to estimate the distribution.

$$P_{MLE}(e = f_i) = Count(f_i) / N \qquad (1)$$

In (1), $fi \in D$, N is the size of sequence, and $Count(fi)$ is the number of occurrences of fi in the sequence.

The proportion of times a certain event f_i occurs is called the relative frequency of the event. In the MLE model, we compute (predict) the relative frequency of each new event based on the preceding sequence. Empirically for our data we observed if one performs a large number of trials, the relative frequency (for each file) tends to stabilize around some number.

In our experiments, instead of definition (1), we computed our MLE probability distributions [1] using this formula:

$$P_{MLE}(E = f_i) = (Count(f_i) + 1) / (N + d) \qquad (2)$$

In (2), $fi \in D$, N is the size of sequence, $Count(fi)$ is the number of occurrences of fi and d is the size of domain D.

We use this equation because equation (1) has two computational problems. The first problem is that it implicitly assigns a zero probability to elements of domain that have not been observed in the sequence. This means it will assign a zero probability to any sequence containing a previously unseen element. The second problem is that it does not distinguish between different levels of certainty based on the amount of evidence we have seen. One solution is to assign a small probability to each possible observation at the start. We do this by adding a small number (we use l) to the count of each outcome to get the estimation formula. This technique, using value 1, is called Laplace estimation [1]. If we never see a token of a type f in a corpus of size N and domain size d, the probability estimate of a token of f occurring will be $1/(N+d)$. For the second problem, using Laplace formula, our prior knowledge that there is D different types of events makes our estimate stay close to the uniform distribution [1].

3.2 Reflexive Exponential Decay (RED) Model

Our second model relies on the idea that when a change is observed in a file, it is likely that more changes will be observed in that file, but that this effect decreases (decays) with time. We are given a sequence of events called $e_1, e_2 \ldots e_n$, occurring respectively at monotonically increasing times $t_1, t_2 \ldots t_n$. We assume that events probabilistically predict events, e.g., bug fixes predict bug fixes. By analogy, yesterday's weather is a good predictor of today's weather.

We postulate that the predictive rate of bugs induced by any event decays exponentially. We call this model *reflexive* because each event in turn predicts more events. More generally, we call it the reflexive exponential decay (RED) model. A particular event occurring at time t_i on the file f_j, implies (predicts) a future frequency rate $R_t(j)$ for that file at future time t. Our model defines $R_t(j)$ as follows:

$$R_t(j) = I\, e^{k(t-t_i)} = I\,(1/2)^{(t-t_i)/h} \qquad (3)$$
$$where \quad k = -\ln(2)/h \ and \ t > t_i$$

In formula (3), h is the half life (measured typically in months) and I is the "impact" of an event (measured typically in events per month). This means that if in the sequence of events, e_i happens at time t_i and e_i is a modification of file f_j, for all time $t > t_i$, the predicted incremental frequency for *file will be* $R_j(t)$.

A larger half life h means that the effects of a change last longer. Figure 5 shows $R_t(j)$ for different half lives and with impact of $I = 1$ and $t_i = 0$.

Based on $R_t(j)$ for each event on file f_j, we define the RED model as the summation of the effects all (historical) events happening to each file.

We now formalize the RED frequency model. Suppose that the sequence of events, $e_0, e_1, e_2, .., e_l$ has happened on file f_j up time t. Then RED predicts that the future frequency of changes to this file will be:

$$R_t(j) = I\, e^{k(t-t_0)} + I\, e^{k(t-t_1)} + I\, e^{k(t-t_2)} + \ldots + I\, e^{k(t-t_l)}$$
$$for\ all\ t \geq t_l \qquad (4)$$

Figure 6 shows how the effect of each event is added to the previous ones for a specific file. In the figure, specific file f_j has been observed to change at times 0, 5 and 15; the individual exponentially decaying predictive effects of these three events are shown as the three lower curves. The cumulative effect of these first two of these (from times 0 and 5) is shown as another curve. Then the effect of all three of these is shown by yet another (the highest) curve.

RED Distribution Model

We will now convert our RED model so that it predicts probability distribution rather than frequency. Given a sequence of events: $e_0, e_1, e_2, .., e_m$, having $R_t(j)$ for all files $j = 1..n$, we can define the distribution of RED at time t as follows:

$$RED_t(e_{m+1} = f_i) = \frac{R_t(i)}{\sum_{j=1..n} R_t(j)} \quad for\ t \geq t_m \qquad (5)$$

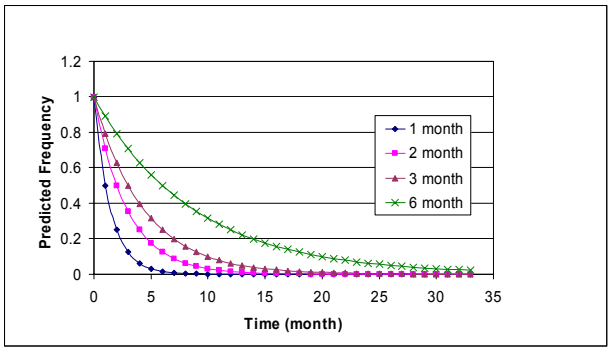


Figure 5. Exponential decay for different half lives.

3.3 RED Co-Change (REDCC) Model

Our third model is an enhanced version of RED. When each event occurs we update the probability not only for the changed file but also for the *co-changed* files. There are several different approaches for concluding that (or defining that) the files change (co-change) together during software development.

Developers commonly modify files together (co-change them) to introduce new features or fix bugs. Developers should ensure that when one file is changed, other related files in the software system are updated to be consistent with the modifications. We use a definition of co-change that is inspired by the literature [8]. If file f_1 and f_2 changed together (on the same day) in previous change sets, then they are candidates to be considered as co-changed files. We will define that co-change files are those sets of files which have changed on the same day in the past at least 3 times within the preceding 7 days. We now define the *RED Co-Change* (REDCC) model. We assume that at time t, the sequence of events, $e_0, e_1, e_2, .., e_m$ has happened on file f_i or on the co-change files of f_i up to this time.

$$REDCC_t(j) = I\,e^{k(t-t_0)} + I\,e^{k(t-t_1)} + \ldots + I\,e^{k(t-t_m)} \qquad (6)$$

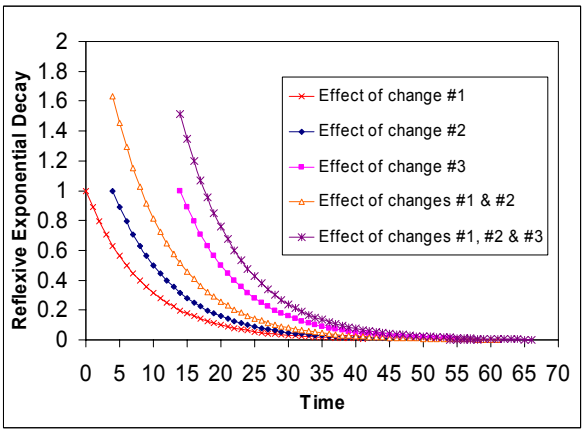


Figure 6. Reflexive exponential decay for a file.

Using $REDCC_t(j)$ frequency model, we convert it to the probability model $REDCC_t(e_{m+1}=f_i)$ in the same way we converted the RED frequency model to a probability model.

4. EVALUATION OF PREDICTION MODELS

In this section we present an information theoretic approach to quantify the goodness or fitness of a guessed probability g (g is a prediction model) compared to the actual probability p. Our approach uses entropy concepts to evaluate prediction models.

Our goal is to compare these predictive models (distributions) to see how good they are. By "good" we mean how close they are to the true distributions of the events. It also could mean that how well they predict the occurrence of the next event. The approach we take is well known in Natural Language Processing (NLP) area, where a sequence of words in language is called *corpus*, but to our knowledge has not been used in the field of Mining Software Repositories. NLP uses information theory to find the distance between prediction models and actual distribution of corpus [11].

4.1 Entropy and Cross Entropy

Before introducing our information theoretic approach, we will review some related concepts. Information theory techniques define the amount of information in a message. The theory measures the amount of uncertainty/entropy in a distribution. Shannon entropy [11], given probability p(x), is defined as:

$$H(p) = -\sum p(x)\log p(x)$$

Larger values of H(p) imply that more bits are needed for coding messages.

There is a related concept called *cross entropy* which allows us to compare two probability functions. (Cross entropy is closely related to Kullback-Leibler divergence [11].) The cross entropy between two probability distributions measures the overall difference between the two distributions p and m and is defined as:

$$H(p,m) = -\sum p(x)\log m(x)$$

Where p(x) is the true distribution and m(x) the model distribution.

The cross entropy is minimal when p and m are identical, in which case it reduces to simply H(p). The closer the cross entropy is to entropy proper, the better m is an approximation of p. If we have two models m_1 and m_2, if $H(p, m_1) < H(p, m_2)$ then m_1 is a closer approximation to distribution to p.

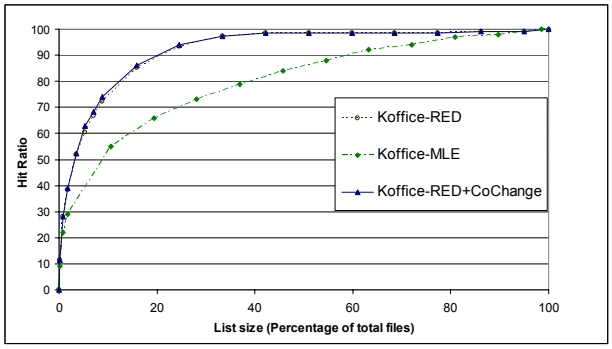
This approach seems to require that we know p, the actual distribution of data, which unfortunately we do not know. One of the central problems we face in using probability models is obtaining the actual distribution p(x) of data. The true distributions are not known, yet we want to estimate predictive models and validate them using the existing data.

Here there is a paradox: if we had p(x) in advance, we wouldn't need to make any model for estimating p(x).

4.2 Corpus Cross Entropy

We solve this problem with using *corpus cross entropy* (CCE). Given a sequence c with of length N consisting of events $e_1 \ldots e_N$, the corpus cross entropy of a probability function m is defined as follows:

$$H_c(m) = -(1/N)\sum \log m(e_i)$$



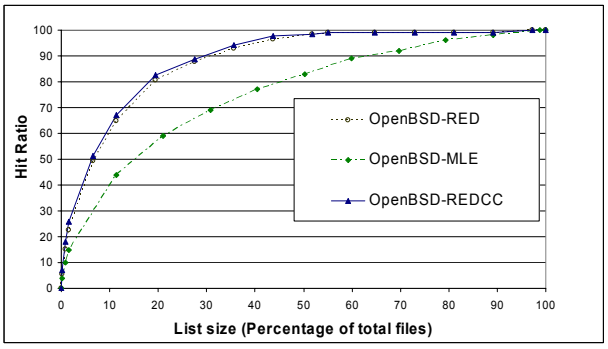


Figure 7. Evaluation of 3 models based on Hit Ratio of Top Ten List, with varying size of list.

It is straightforward to prove that corpus cross entropy $H_c(m)$ approaches cross entropy H(p,m) as N approaches infinity, given that p is the true distribution of corpus c and given that p is stationary. We can compute $H_c(m)$, as an approximation to H(p,m), even though we do not know distribution p. As is done in NPL literature [11], we assume that given two models m_1 and m_2 we can compare $H_c(m_1)$ and $H_c(m_2)$ to determine which of m_1 and m_2 is the better model, even though we do not know the true distribution p, given that p is reasonably stationary. That is, when $H_c(m_1) < H_c(m_2)$ we conclude that m_1 is a closer distribution to the true distribution and hence is a better model.

5. EMPIRICAL STUDIES

In this section we evaluate our three proposed prediction models (MLE, RED and REDCC) empirically, using two approaches, for six large open source systems. Table 1 summarizes the details of the software systems we studied. Due to space limitation we will only shown the results for two systems (Koffice and NetBSD). The other systems had similar behavior.

5.1 Top Ten List evaluation

For evaluating the quality of our three models, first we use the Top Ten List [8] approach. This approach evaluates which model predicts more accurately.

In this approach, the model predicts a list of the 10 files (more generally, a list of *n* files) that are most likely to be changed next. A new list is generated for each new event.

Given a predicted distribution m for the next event, we create the corresponding Top Ten List for that upcoming event by picking the ten (or *n*) files with the highest probability according to m.

With the occurrence of each event, there is a change to a file, call it file f_i. We record whether file f_i is in the event's Top Ten List. We define the Hit Ratio as the fraction of events in which file f_i was observed to be in its Top Ten List. Models with higher Hit Ratios are considered to be better models.

We applied the Top Ten List approach to evaluate our three proposed models, for all the studied system; see Figure 7 for the results for two of these systems: Koffice and OpenBSD. (Results for the other studied systems are comparable.) As can be seen, for both systems, REDCC and RED have very similar results, with REDCC being slightly superior. By contrast, MLE's results are considerably worse. In other words, the Top Ten List approach evaluates REDCC is slightly better than RED, and both of these considerable better than MLE.

As can be seen in Figure 7, as the size of list increases, we have a higher hit ratio. Interestingly, using REDCC or RED model, when we use 20 percent of total files in the system, the hit ratio is almost 80 percent.

5.2 Information theoretic evaluation

We also applied the information theoretic approach to compare our three prediction models. Due to space limitations, we only present the result for one of the studied system, Postgres. The results for the other systems are similar.

Using historical Postgres data, we developed instances of our three models: MLE, RED and REDCC. To develop the MLE model, we used the first 10000 events and kept it fixed for the remaining corpus.

Figure 8 shows the corpus cross entropy of our three predictive models when applied to Postgres. As it can be seen in the figure, REDCC has the lowest corpus cross entropy which means its distribution is the closest to the actual distribution of the data. The next closest (see middle curve in Figure 9) is RED, and the worst (top curve) is MLE. Note that this ordering is the same that we observed when our evaluations were based on the Top Ten List.

As can be seen in Figure 8, as the size of corpus increases the MLE distribution gets farther from the real distribution of data but for two other models, RED and REDCC, the opposite is true. This suggests that the RED and REDCC models benefit by updating their distributions based on the events in the corpus as time passes.

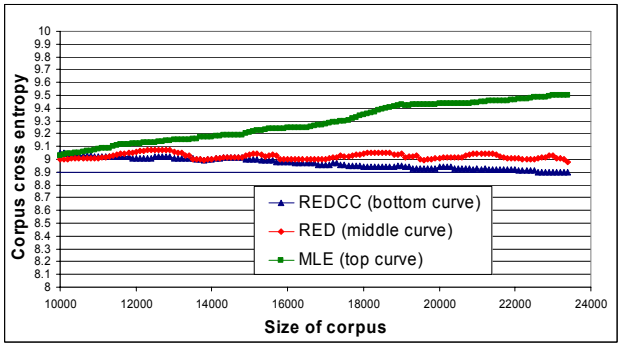


Figure 8. Evaluation of 3 models using corpus cross entropy on Postgres.

6. CONCLUSION

We developed three models (MLE, RED and REDCC) for predicting future modification of files based on available change histories of software. We proposed a rigorous approach for evaluating such predictive models. This approach has been used in Natural Language Processing, but not in Mining Software Repositories, as far as we know. This is an information theoretic approach in that the closeness of a predictive model distribution to an actual but unknown probability distribution of the system is measured using cross entropy. We evaluated our proposed prediction models empirically using two approaches for six large open source systems. First we used the Top Ten List [8] approach to see which model predicts more accurately. Using this approach we showed that the REDCC model works best of our three models. Then using our information theoretic evaluation approach, we observe that the REDCC model again has the distribution that is closest to the actual distribution for all the studied systems. An advantage of our information theoretic approach over the Top Ten List approach is that using our approach we know quantitatively, as measured by cross entropy, how much better or worse is the prediction model compared to ideal result.

Our hope is that our approach can be used to help better predict future changes and bugs, based on the history of software. Our approach also can be used by researchers who have developed new prediction models to evaluate them using a information theoretic approach.

7. ACKNOWLEDGMENT

The authors would like to thank Ahmed Hassan. This paper would not have been possible without his generous help and his data. We also would like to thank the referees for their extremely helpful suggestions.

8. REFERENCES

[1] Allen, J. F. Using Entropy for Evaluating and Comparing Probability Distributions, available at: http://www.cs.rochester.edu/u/james/CSC248/Lec6.pdf

[2] Basili, V. R., and Perricone, B. Software errors and complexity: An empirical investigation. Communications of the ACM, 27(1):42 – 52, 1984.

[3] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J.S., and Mockus, A. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. on Software Engineering,* 27(1):1–12, 2001.

[4] Eick, S.G., Graves, T.L., Karr, A.F., Mockus, A., Schuster, P. Visualizing Software Changes, *IEEE Trans. on Software Engineering*, vol. 28, no. 4, pp. 396-412, April, 2002.

[5] Gall, H., Hajek, K., and Jazayeri, M. Detection of logical coupling based on product release history. *In Proceedings of the 14th International Conference on Software Maintenance,* Bethesda, Washington D.C., November 1998.

[6] Graves, T. L., Karr, A. F., Marron, J. S. and Siy, H. P. Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering,* 26(7):653–661, 2000.

[7] Hassan, A. E., *Mining Software Repositories to Assist Developers and Support Managers*. PhD Thesis, University of Waterloo, Ontario, Canada, 2004

[8] Hassan, A. E. and Holt, R. C., The Top Ten List: Dynamic Fault Prediction, *Proceedings of ICSM 2005: International Conference on Software Maintenance,* Budapest, Hungary, Sept 25-30, 2005.

[9] Khoshgoftaar, T. M., Allen, E. B., Halstead, R., Trio, G. P. and Flass, R. M. Using Process History to Predict Software Quality. *Computer,* 31(4), 1998.

[10] Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., and Hudepohl, J. P. Data Mining for Predictors of Software Quality. *International Journal of Software Engineering and Knowledge Engineering,* 9(5), 1999.

[11] Manning, C. and Schütze, H. Foundations of Statistical Natural Language Processing, MIT Press. Cambridge, MA: May 1999.

[12] Mockus, A. and Votta, L. G. Identifying reasons for software change using historic databases. In *International Conference on Software Maintenance*, pages 120-130, San Jose, California, October 11-14 2000

[13] Mockus, A., Weiss, D. M., and Zhang, Ping. Understanding and predicting effort in software projects. In *2003 International Conference on Software Engineering*, pages 274-284, Portland, Oregon, May 3-10 2003. ACM Press.

[14] Ostrand, T. J., Weyuker, E. J., Bell, R. M. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Trans. Software Eng.* 31(4): 340-355 (2005)

[15] Pareto Law: http://www.it-cortex.com/Pareto_law.htm

[16] Perry, D. E. and Evangelist, W. M. An Empirical Study of Software Interface Faults — An Update. *In Proceedings of the 20th Annual Hawaii International Conference on Systems Sciences,* pages 113–136, Hawaii, USA, January 1987.

[17] Perry, D. E. and Steig, C.S. Software Faults in Evolving a Large, Real-Time System: a Case Study'. *In Proceedings of the 4th European Software Engineering Conference,* Garmisch, Germany, September 1993.

[18] Reliability Analysis Center, Introduction to Software Reliability: A state of the Art Review. Reliability Analysis Center (RAC), 1996. http://rome.iitri.com/RAC/

[19] Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A. Mining Version Histories to Guide Software Changes, *IEEE Trans. on Software Engineering*, vol. 31, no. 6, pp. 429-445, June, 2005.

[20] Zipf, G. K. Human Behavior and the Principle of Least Effort.Addison-Wesley, 1949.

Tracking Defect Warnings Across Versions

Jaime Spacco*, David Hovemeyer†, William Pugh*

*Dept. of Computer Science
A. V. Williams Building
University of Maryland
College Park, MD 20742 USA

{jspacco,pugh}@cs.umd.edu

†Dept. of Computer Science
Vassar College
124 Raymond Ave.
Poughkeepsie, NY 12604 USA

hovemeyer@cs.vassar.edu

ABSTRACT

Various static analysis tools will analyze a software artifact in order to identify potential defects, such as misused APIs, race conditions and deadlocks, and security vulnerabilities. For a number of reasons, it is important to be able to track the occurrence of each potential defect over multiple versions of a software artifact under study: in other words, to determine when warnings reported in multiple versions of the software all correspond the same underlying issue. One motivation for this capability is to remember decisions about code that has been reviewed and found to be safe despite the occurrence of a warning. Another motivation is constructing warning deltas between versions, showing which warnings are new, which have persisted, and which have disappeared. This allows reviewers to focus their efforts on inspecting new warnings. Finally, tracking warnings through a series of software versions reveals where potential defects are introduced and fixed, and how long they persist, exposing interesting trends and patterns.

We will discuss two different techniques we have implemented in FindBugs (a static analysis tool to find bugs in Java programs) for tracking defects across versions, discuss their relative merits and how they can be incorporated into the software development process, and discuss the results of tracking defect warnings across Sun's Java runtime library.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics, Symbolic Execution; D.2.2 [**Design Tools and Techniques**]: Programmer workbench

General Terms

Human Factors, Languages, Verification

Keywords

FindBugs, Java, bug histories, bug tracking, static analysis

1. INTRODUCTION

There are many tools that perform static analysis of software to detect possible software defects. Each of these tools looks for some mixture of security vulnerabilities, coding errors, poor programming practice and style violations.

It is naive to assume that after software is analyzed, the software will be immediately modified to eliminate all of the warnings generated by the static analysis tool. Much more typically, developers will choose, for some reason or another, to change the code in response to only a some of the warnings. Even if a static analysis tool is based on precise and sound reasoning, warnings have to compete with other demands on the developers time, and minor problems may not be worth fixing if there is a chance that a change may introduce an incompatibility or some new, more serious defect.

Thus, when the next version of the software is built and analyzed, many of the warnings will reflect issues from the previous version that were not addressed, while other warnings will correspond to new issues. Being able to pair up warnings generated from analyzing different builds of software is a vitally important task. It is not particularly difficult, but it has not been studied or discussed at length in the literature. In our work on the FindBugs static analysis tool [5, 3] we have implemented techniques and tools for tracking warnings across versions. In recent discussions with Fortify Software [4] we found that their approach to the problem is substantially different than the one we used in FindBugs. We have now implemented both approaches within FindBugs, and in this paper we describe them and report on their relative strengths and weaknesses.

2. THE PROBLEM

First, we need to identify the problem we wish to solve a little more precisely. There are actually a number of similar use cases, which are all largely addressed using the same techniques:

- Assume that a particular version of software was analyzed, generating a list of warnings about potential defects. Some of these warnings were audited, with some of them being flagged as harmless and others being flagged as serious problems (but perhaps not yet fixed). When a new version of the software is analyzed, we want to be able to associate the audit results from the previous analysis with the issues raised by analyzing the current version of the software. Thus, we can ignore the issues previously marked as harmless, and ensure that the ones previously marked as important continue to be flagged as important.
- A similar problem, except more decentralized. The development team might currently be working on build b55, while at the same time one security team is auditing build b48 of the servlet library, and another security team is auditing b50 of the persistence library. How do the two security teams relay their findings to the development team?
- A development team has just started using static analysis tools,

MSR'06, May 22–23, 2006, Shanghai, China.

and the tool generates more than 300 serious warnings. The team doesn't have time to review all of the issues, so they want to review only the new warnings—those that did not occur in previous versions of the software.

3. MATCHING WARNINGS IN FINDBUGS

3.1 Pairing

The first and primary form of matching implemented in FindBugs is based on pairing warnings. We start with two sets of warnings. We then try to match up warnings with a progressively "fuzzier" series of WarningMatchers. Each matching object provides a hash function and equivalence predicate for warnings, with a property that for any WarningMatcher m, and warnings w1 and w2, m.equivalent(w1,w2) implies m.hashCode(w1) = m.hashCode(w2). We first start with a very precise warning matcher, which only considers two warnings the same if all recorded details about them are identical.[1]

The first matcher will generally pair up and remove from consideration that vast majority of warnings. We then apply fuzzier matchers, which allow for source lines to vary. If there are collisions (e.g., two warnings from version A and two warnings from version B both match), we pair them up according to their lexicographical order in the warning database, which is determined by the lexicographical order of named elements such as classes, fields, and methods, and by the order of the byte code offsets of any source line references.

As we move to even fuzzier matching algorithms, we look for package renaming. If, in one version, there is a warning in class org.apache.Foo, and in the next version there are no classes in the org.apache package, but there is a new package com.sun.org.apache containing a Foo class, we consider that to be a package renaming and allow warnings in the org.apache.Foo class to be matched to com.sun.org.apache.Foo.

At the moment, we do not try to accommodate refactorings that would move a bug warning from one method to another, even in trivial cases such as renaming a method.

3.2 Warning signatures

A second approach to matching warnings is *warning signatures*. For warning signatures, we compute, for each warning, a string including the names of classes, fields, and methods involved in the warning, but excluding source locations. We then compute the MD5 hash of the string and represent the hash value in hexadecimal so that all warning signatures are a consistent length.

The significant problem here is how to handle collisions: two different warnings producing the same MD5 hash. We do not expect actual MD5 collisions—different strings producing the same MD5 hash—to be an issue. Rather, the problem is what if, for example, the tool finds two possible SQL injections in the same method, such that everything other than the source line offsets (which are ignored by the signature computation) are identical?

As of version 3.5, Fortify Software used a similar mechanism for computing warning signatures, but upon collision they simply rehashed the signature: in other words, computing the MD5 hash of the original MD5 hash. We felt that this was a bad choice, since it makes it exceedingly difficult to recover information about where potential collisions occurred.

In our first implementation of warning signatures in FindBugs (available in FindBugs version 0.9.5), we compute occurrence numbers for each warning. Assuming there are no collisions, each warning gets an occurrence number of zero. If there are collisions, successive occurrence numbers are generated (in order of byte code offset for the first source line annotation). Thus, by concatenating the warning signature and the occurrence number, we get a string that is guaranteed to be unique for any collection of warnings. When matching warning signatures across versions, we know that any warnings with a non-zero occurrence number indicate a collision, and possible mismatching of warnings.

In reviewing the places where FindBugs can generate multiple warnings per method, we found that in many cases the warnings were all related, and it made sense to only report one such warning per method. Thus, a number of bug pattern detectors were changed so that they would report at most once per method, but each warning would contain a list of all the source lines where the issue arose. This allows correct matching without collisions, even if the number sub-issues or their source line offsets change between versions.

[1]Any source locations here are denoted by bytecode offsets rather than source lines, because bytecode offsets are not affected by changes to methods elsewhere in the file.

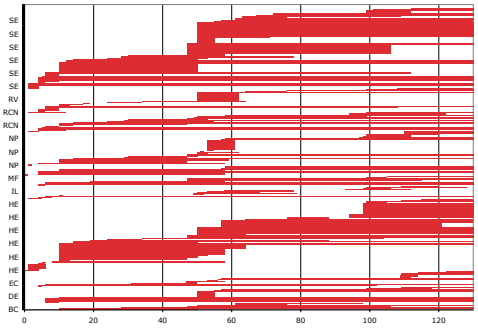


Figure 1: Lifetimes of high priority correctness warnings in Sun's JDK

4. RESULTS OF TRACKING DEFECTS USING FINDBUGS

4.1 Sun's JDK

We have a reasonably complete history of the core runtime library (rt.jar) from releases of the Sun Java Development Kit (JDK), including 116 sequential builds, starting with release 1.0.2, and including bi-weekly or weekly beta builds of the 1.5.0 and 1.6.0 JDKs. We analyzed the longest possible sequence of versions where both release date and version number increased monotonically: in other words, once we analyzed b12 of 1.6.0, the first publicly released build, we didn't analyze any later builds in the 1.5 branch.

4.1.1 Warning lifetimes

Figure 1 shows the lifetimes of all the high priority correctness warnings across all versions of the JDK that we analyzed, excluding 5 other miscellaneous warning types that did not fall into any category large enough to depict in the figure. The ticks on the x-axis correspond to successive builds of the JDK. Each horizontal line corresponds to one warning and stretches from the build where the warning was first detected to the last build in which it existed. The defect warnings are grouped by FindBugs bug type (e.g., IL is an infinite recursive loop).

There are several interesting things to note about this diagram. First, there are builds where bulk changes occur; where many defects are introduced or fixed. The JDK development process includes a fair bit of parallel development, and many of these places represent

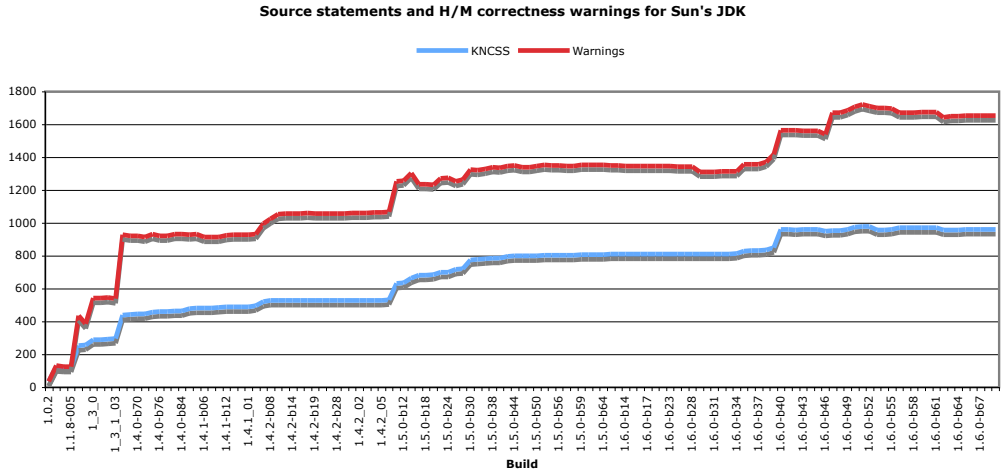


Figure 2: Code size and number of defect warnings in Sun's JDK

places where updates to particular packages are being integrated into the main branch.

Also, the majority of the high-priority defects we found using FindBugs have been fixed. We found one bug pattern, infinite recursive loops, to be so compelling and amusing that we filed bug reports on all of the infinite recursive loops we found. All of those have been fixed, save one. That bug originated in build 1.3.0, more than 5 years ago, and we believe the reason that bug remains unfixed is that the code is stable without an active group developing or maintaining it.

4.1.2 *Code size and defect density*

Figure 2 shows the size of the JDK builds over time, and the number of both medium and high priority correctness warnings existing in each build.

The size of the JDK builds is given in thousands of non commenting source statements. For most classes, this is computed from the table associated with each method that maps byte code offsets to source line number. Since there is only one entry per statement, this correctly handles whitespace and statements that spread over several lines. In the few cases where we analyze classfiles that do not contain line number tables, we extrapolate from the empirically observed value of 10 bytecodes per non commenting source statement.

Over time, we observe a warning density that grows from about 1 warning per KNCSS in early builds to 2 warnings per KNCSS. However, this does not necessarily reflect that the quality of the JDK codebase has decreased over time. Rather, the warnings in a build correspond both to unfixed defects and false positives. Because false positives accumulate over time (since they do not warrant corrective action to the code) we would expect that the total combined density of false positives and defects would grow over time, even if the defect density remains constant.

4.1.3 *Defect warning decay over time*

In Figure 3, we show the number of correctness warnings in each build of the JDK that satisfies the following:

- The defect was first reported before 1.4 builds
- The defect did not disappear because the class that contained it disappeared
- The defect was not a warning about classes that are not serializable.

The warnings about non-serializable classes were excluded since a systematic effort was made at Sun to add serialVersionUID fields to all the classes that might need them; this resolved several hundred medium priority issues and would otherwise swamp the results. We exclude the defects in removed classes because their removal gives us very little information about why the warning was removed.

From this, we can see that over time, more than half of the high and medium priority correctness warnings are removed. The fact that a lesser portion of the low priority warnings are removed over time gives us reason to believe that this is due to reasons other than code churn, and that the issues we identify as high and medium priority correctness issues are more likely to be things that developers independently determine to need fixing than low priority warnings.

4.1.4 *The java.util experience*

The java.util package, which contains various classes such as the Collections libraries, is some of the most carefully scrutinized Java code in Sun's JDK implementation. It has been widely reviewed, and has largely been written by two developers, Joshua Bloch and Martin Buchholz, who are highly skilled and highly dedicated to getting their code correct. They also use and advocate the use of the FindBugs tools, and there has been substantial discussion between the FindBugs team and the maintainers of the java.util package. We file a bug report on every defect that FindBugs finds in the java.util package, and also examine every false positive generated by FindBugs on this package. We have not tuned FindBugs to specifically exclude any false positives we might generate on java.util, although we do look for reasons we report false positives in java.util that might be more widely applicable.

Given this background, it is useful to see how FindBugs performs on the java.util package. In the latest build of JDK 1.6.0 (build 69), the java.util package consists of 273 classes and 18,765 noncommenting source statements; using wc to total lines in source files gives 59,175 source lines. As of build 1.6.0-b69, FindBugs identifies 4 warnings in java.util. Of these, one is a real and serious defect which will be fixed before the 1.6 release. The remaining 3 are false positives. FindBugs generates 23 warnings in previous versions of the java.util that are no longer generated in the current version. These results suggest both

- in comparing the density of both active and dead defect warnings, the java.util package has a defect density less than half of that of the JDK overall,

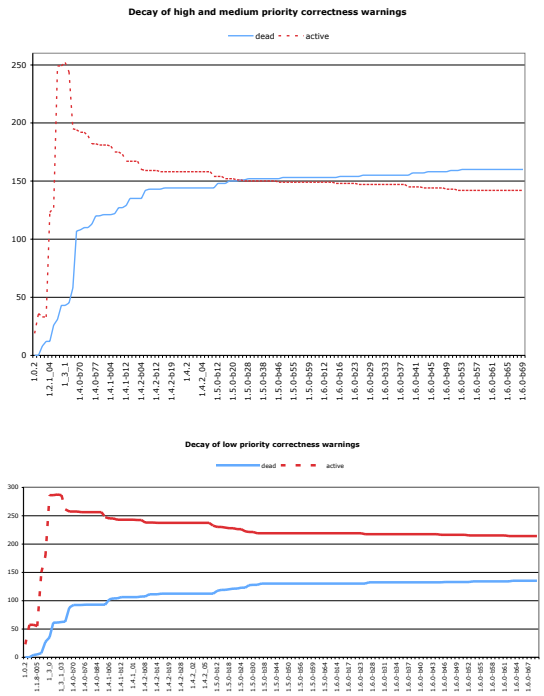


Figure 3: Decay over time of defect warnings introduced before 1.4 builds

- with more attention to improving code quality and improving FindBugs, the ratio that 8 out of 9 of the issues identified by FindBugs are ones that developers believe should be fixed should be more widely reproducible.

5. SOFTWARE EVOLUTION

Two common software engineering practices, creating or merging branches in a repository and renaming packages, create difficulties that the two warning-matching techniques handle differently.

5.1 Non-linear branches

As software evolves, developers need to branch repositories into separate development streams, or merge separate repositories into a single development stream. For example, a version of the JDK-1.4 was branched to provide a starting point for the development of JDK-1.5, while Doug Lea's util.concurrent library was merged into the JDK as java.util.concurrent.

However, after a branch for a new version is created, maintenance on the branch for the old version continues in parallel for some amount of time, often months or year. Similarly, maintenance can continue on a module after it is merged into another repository.

In this environment, a developer who encounters a bug warning needs to know if the same issue occurs in any other branches of the development process, because the issue may already have been fixed or marked as a false positive on a different branch.

The two techniques for matching warnings, *pairing warnings* and *warning signatures*, handle this problem differently, with advantages and disadvantages to each approach.

The algorithm for pairing warnings used by FindBugs was designed with a linear sequence of software versions in mind. Pairing warnings is more fine-grained than warning-signatures in that it can determine, for example, which potential null-pointer dereference in a method was fixed between two versions. This type of fine-grained information is very useful to a developer actively working on a linear branch who needs detailed information about bug warnings in order to best focus his resources.

However, the pairing implementation currently assumes that the lifespan of a warning is defined by the version in which it is introduced and the last version in which it still exists; there is currently no support for a warning with a set of disjoint lifespans. Thus, as it is currently implemented it would be difficult to use the pairing approach to match warnings between branches of software. In the future we hope to improve the matching algorithm to address this limitation.

The warning-signatures approach used by Fortify Software computes a unique hash for each instance of a warning based on a string representation of the name of the bug pattern and the name of the method and classfile in which it occurs. If there are collisions, the string value is simply re-hashed to resolve the collision. A major benefit is that the hash can easily be used to find warnings in earlier versions of a linear development stream as well as in separate branches of parallel development.

The major drawback to this approach is that collisions cause the analysis to lose information about which bugs are fixed. Because the hashes don't take into account line number information or bytecode offsets, there is no way to determine which bug warning was removed between versions of code: it will always appear as if the second (re-hashed) value was removed.

Thus, the warning-signatures approach is more appropriate for developers who need to fix bugs across branches of software, or who need to integrate branches of software together.

6. CONCLUSION

The issue of matching warnings between versions has only recently been addressed, perhaps because many people assumed the problem was easy and focused their attention on building new bug detectors instead. However, as we've illustrated in this paper, matching warnings is a tricky, interesting problem that needs to be properly addressed for static error checkers to find their way into mainstream software design. In addition to the obvious practical applications (false positive suppression, applying audit results between versions, and construction of warning deltas), studying the lifecycle of defect warnings provides an interesting new perspective on code evolution.

The lack of previous attention paid to the issue is reflected by the fact that several approaches have appeared, each with its own relative advantages and disadvantages, but nobody has studied the various approaches or tried to unify them into a common framework that leverages the advantages of each approach.

7. RELATED WORK

A recent thread on Slashdot [1] discusses the difficulties involved in handling bug reports across branches for bug reporting systems such as Bugzilla [2]. However, we are not aware of published work on this subject.

8. REFERENCES

[1] Bug tracking across multiple code streams? `http://ask.slashdot.org/article.pl?sid=05/10/06/2248259&tid=128`, 2006.

[2] bugzilla.org. `http://www.bugzilla.org/`, 2006.

[3] FindBugs—Find Bugs in Java Programs. `http://findbugs.sourceforge.net`, 2006.

[4] Fortify Software. `http://www.fortifysoftware.com`, 2006.

[5] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Companion of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, October 2004.

Mining Email Social Networks*

Christian Bird, Alex Gourley,
Prem Devanbu, Michael Gertz
Dept. of Computer Science, Kemper Hall,
University of California, Davis,
Davis, California Republic.
cabird,devanbu@ucdavis.edu

Anand Swaminathan
Graduate School of Management,
University of California, Davis,
Davis, California Republic.
aswaminathan@ucdavis.edu

ABSTRACT

Communication & Co-ordination activities are central to large software projects, but are difficult to observe and study in traditional (closed-source, commercial) settings because of the prevalence of informal, direct communication modes. OSS projects, on the other hand, use the internet as the communication medium, and typically conduct discussions in an open, public manner. As a result, the email archives of OSS projects provide a useful trace of the communication and co-ordination activities of the participants. However, there are various challenges that must be addressed before this data can be effectively mined. Once this is done, we can construct social networks of email correspondents, and begin to address some interesting questions. These include questions relating to participation in the email; the social status of different types of OSS participants; the relationship of email activity and commit activity (in the CVS repositories) and the relationship of social status with commit activity. In this paper, we begin with a discussion of our infrastructure and then discuss our approach to mining the email archives; and finally we present some preliminary results from our data analysis.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Empirical, Open Source*

General Terms

Human Factors, Measurement

Keywords

Open Source, Social Networks

*We gratefully acknowledge support from NSF Humanities and Social Sciences Division, Grant Number SES 0525263.

MSR'06, May 22–23, 2006, Shanghai, China.

1. INTRODUCTION

Large-scale software development projects invariably require a lot of communication and coordination (C&C) amongst the project workers. We distinguish these activities from engineering activities, where actual artifacts such as source code or documents are modified. The difficulty and intensity of the required coordination effort is quite high; this is often cited as the reason why adding more developers doesn't necessarily speed-up development [4]. C&C activities influence (and are influenced by) the design, structure and evolution of software systems. In traditional, commercial software organization, C&C activities may occur informally, and would be difficult to study. Even if coordination and communication are computer-mediated, the traces of these activities are usually not made public by commercial organizations. Open-source software (OSS) projects on the other hand, inherently conduct all their activities in public, and in fact, this public, open enactment is key to their success [16, 11]. In particular, every open-source project includes one or more public mailing lists wherein project stakeholders can communicate and coordinate their activities. The entire trace of these mailing lists are archived and available for study.

These archives, along with the versioned source code repositories and other on-line artifacts constitute a unique and valuable resource for the study of C&C activities in software projects. There is at UC Davis an interdisciplinary effort to mine this resource, and use the resulting data to study the relationship with C&C activities in OSS projects, and the actual development activities. In this paper, we describe our experiences with this effort, and some early results. We begin first with a description of the phenomena that we are mining; then we describe our data extraction tools; finally, we present an early look at the data.

2. CHATTERERS & CHANGERS

A mailing list in an OSS project is a public forum. Anyone can post messages to the list. Posted messages are visible to all the mailing list subscribers. Posters to mailing lists include developers, bug-reporters, contributors (who submit patches, but don't have commit privileges) and ordinary users. Mailing lists can be quite active; for example, on the Apache developer mailing list, there were about 4996 messages in the year 2004 and 2340 in 2005. For gcc, these numbers were 19173 and 15082. Over the lifetime of the project, we estimate that over 2000 distinct individuals have sent messages to the Apache developer list. A subscriber may

respond to a message on the public forum, which then becomes visible to everyone. Roughly 73% of messages elicit response messages. A response b to a message a is an indication that the sender of $b, (s_b)$ found that the sender of $a, (s_a)$ had something interesting to say; thus the response from s_b indicates that the original message a represented information flowing from s_a to s_b It is also an indication of status, *i.e.,* s_b indicates that s/he found s_a's email worth reading, and worthy of response.

The level of activity of developers on the mailing list varies dramatically. The most active developer on the mailing list sent 4486 messages during the life of the project. The least "chatty" developer sent just 10 messages. There were, of course, non-developers who sent just one message. Messages reflect communication interactions between developers. Some developers have a great many interactions: one developer's emails had responses from 254 distinct individuals. Likewise, another developer replied to messages from 281 distinct individuals. However, the vast majority of individuals particpating on the email list sent very few messages, and received very few replies to their messages. This type of "Pareto" distribution is common in social phenomena [14].

The community on the Apache developer mailing list is concerned primarily with software, and so the question naturally arises as how email activity relates with development activity. This activity can be conveniently recovered from the versioned source code repository (CVS in this case). As has been reported in earlier research [8] on Linux, development activity, as recorded in CVS, also shows a few developers doing the bulk of the work.

Our research goal is to study the relationship of the C&C activities of developers, as revealed in the email archives, to their software development activity. Specifically, we are interested in how the activities and connections between developers on the mailing list relate to their development activity in the source code. We are interested in the following types of questions:

- *What are the properties of the social network of developers?*
- *Are developers who send a lot of messages on the mailing list also very active in source code changes?*
- *Do developers play a different role than non-developers in the social network?*
- *Do the most active developers have the highest status among developers ?*

Unfortunately, answering these types of questions requires facing some challenges in data extraction, primarily having to do with resolving aliasing issues on the email archives and cvs archives.

3. OF DOGS AND DEVELOPERS

"On the Internet, no one knows if you're a Dog" —so goes the famous New Yorker Cartoon. It is difficult (and sometimes impossible) to determine the identity of individuals who correspond on mailing lists using aliases. The same individual can use different email aliases. For example the developer *Ian Holsman* uses 7 different email aliases, including `ian.holsman@cnet.com`, `ianh@holsman.net`, and `ianh@apache.org`[1]. Sometimes aliases have very little relationships to developers (or dogs): the developer *Ken Coar* uses the name *Rodent of unusual size* associated with email address `ken.coar@golux.com`. Ignoring these aliases and treating these as distinct email personalities would confound later steps of data analysis. Likewise, when cvs comments are made, developers use a cvs account name. Fortunately, since access and accounts to cvs are controlled centrally, there is less of an aliasing problem with cvs account names. However, in order to relate email activity and programming/development activity, we must correlate email names with cvs account names. Given the possibility of choosing arbitrary aliases, one can make two important observations: first, an individual determined to maintain an anonymous alias can always do so[2] ; second, any automated algorithm for resolving aliases will be inexact, and must be supplemented by subsequent manual analyses.

We now describe our hybrid automated/manual approach to resolving aliases

3.1 Unmasking Aliases

Most emails include a header that identifies the sender, of this form:

```
From: "Bill Stoddard" <reddrum@attglobal.net>
```

This header reveals immediately the problem—*Bill Stoddard*, who here uses the handle `reddrum` is actually also `bill@wstoddard.com`. But how can we know that?

Overview: Our first step in resolving aliases is to automatically crawl messages and extract all such headers to produce a list of $< name, email >$ identifiers (IDs). Once this is done, we execute a clustering algorithm that measures the similarity between every pair of IDs. This could occur if either the names are similar, or if the emails are similar, or if the names and the emails are similar (the precise details of the algorithm are explained below). IDs that are sufficiently similar are placed into the same cluster. Once clusters are formed, they are manuall post-processed.

Apache Summary: In the case of the Apache developer mailing list, we began with 2544 separate IDs. The clustering algorithm produced 1581 clusters. The largest of these had 70 members, the next biggest 55, and so on; finally, there were 163 doubles, and 1271 singletons. Naturally, these clusters contained errors, and had to be manually post-processed. Mindful of the need for manual post-processing, we deliberately set the cluster similarity threshold quite low: it is much easier during a manual step to split clusters than to unify two disparate clusters from a very large set. Manual processing of the 1581 clusters produced 2012 distinct individuals, some of whom have many aliases. One noteworthy example is *Rasmus Lerdorf*, with 11 aliases:

```
rasmus@apache.org,
rasmus@bellglobal.com,
rasmus@lerdorf.ca,
rasmus@lerdorf.com,
rasmus@lerdorf.on.ca,
rasmus@linuxcare.com,
```

[1]Email addresses are used with permission from the mailinglist participants.

[2]The identity of the infamous *"David who wishes to remain anonymous"*, who spammed several email lists, offering to post personal adverts in Ukraine, was not easily found.

rasmus@madhaus.utcs.utoronto.ca,
rasmus@mail1.bellglobal.com,
rasmus@php.net,
rasmus@raleigh.ibm.com,
rasmus@vex.net.

Five of these were discovered using the *Name Similarity* rule (described below); the other six were put into this cluster because of the *Email similarity* rule. *Paul Richards* was another promiscuous email masquerader, with 10 aliases. Subsequent random sampling of the clusters by one of the authors (who didn't do the manual filtering) revealed no discernible errors (see caveat on this later).

Clustering Algorithm: This algorithm takes a flat list of IDs and clusters them (recall that an ID is a $< name, email >$ tuple). The first step is create pairwise similarity measures for every pair of IDs. Two IDs with similarity measure exceeding an empirically set threshold are placed into the same cluster. The similarity measure is computed by proceeding as follows:

1. *Normalize name:* We remove all punctuation, suffixes (*"jr"*); turn all whitespace into a single space; remove generic terms like "admin", "support", from the name; we also split the name (using whitespace and commas as cues) into first name and last name.

2. *Name Similarity:* We use a scoring algorithm based on the Levenshtein edit distance [5, 13, 17] between the full names, and the first and last names separately. We consider names similar if the full names are similar, or if both first and last names are similar. Thus, *Andy Smith* is similar to *Andrew Smith*, but *Deepa Patel* is dissimilar to *Deepa Ratnaswamy*. This is a very productive rule for identifying clusters of similar emails.

3. *Names-email Similarity:* Two IDs are also scored highly similar if the emails and names match. If the email contains both first and last names (and the lengths of the names are at least 2 characters) we consider them matched. Also, if the email contains the initial of one part of the name and entirety of the other part, then it is considered a match. Thus *Erin Bird* matches `erinb` and `ebird`.

4. *Email Similarity:* If the Levenshtein edit distance between two email address bases (not including the domain, after the "@") is small, two emails are considered similar (as long as the two bases at least 3 characters long)

5. *Cumulative ID similarity:* The similarity between two IDs is the maximum of the 3 mentioned above. This generous rule creates larger clusters; however, splitting too-large clusters is easier than unifying smaller clusters (from a very large number of clusters).

The Cumulative ID similarity is computed for all pairs of IDs; IDs with similarity exceeding a threshold are placed into clusters. The clusters are then manually post-processed as described above. The final results produced were hand-inspected by another member of the team, and appears to be free of evident errors. Of course, given the possibility of choosing arbitrary aliases, such manual inspection is fallible. In future work, we will undertake a more formal, sampling-based techniques technique for determining bounds on the error rates in our results. We propose to email a randomly chosen subset of individuals on the list, and ask them if the set of aliases we have found for them is accurate. Assuming that mis-classification errors are uniformly distributed in our clusters, we should be able to calculate confidence intervals on the actual error rate in our clusters.

CVS alias resolution: We use a similar approach to resolving cvs account names to email aliases. Similarity metrics are calculated on all pairs of mailing list aliases and CVS names. The final matched list is hand-inspected, also as described above; the same caveats apply, and in future work, we will use the same random sampling approach to statistically bound the errors in our results.

3.2 Data Extraction

We gathered data by parsing the email activity on the Apache HTTP Server Developer mailing list over a period starting in 1999 to the current date. Earlier email data was not included because we do not have version-control information before then; we only used the email data for the period during which the source code change data was also available. For every email, we extracted from the email header the message identifier, the sender, the sent time, and the identifier of the message (if any) to which this message was a reply. When a reply-to header was found, the sender s of the reply was someone who found the initial message of interest; and so the sender s was marked as a recipient of the original message. In this way, we were able to extract communication links between pairs of individuals.

We were able to parse 101,637 messages out of 102,611 messages in the mailing list. A small proportion of messages could not be parsed, because of malformed headers. Approximately 1.3% of the messages were in this category. Malformed headers can fail to provide a message identifier, and can also fail to provide a reply-to identifier. This could be due to misbehaving email clients. We are working on ways to rectify this problem. Quoted text content (as done in [1]) is one approach, whereby one message is identified as a reply if it quotes text from another. Meanwhile, we believe that our results are reasonably robust, and would not be affected much when these (currently unparseable) messages are included in the analysis.

4. DESCRIPTION: SMALL WORLD

The distributions of the data that are shown in Figure 1 describe the behavior of the participants of the email list. Each is a histogram showing the number of people exhibiting a particular kind of behavior. The character of the distributions is consistent with previously observed social phenomena, and show the typical long-tailed characteristic in the log-log domain plot.

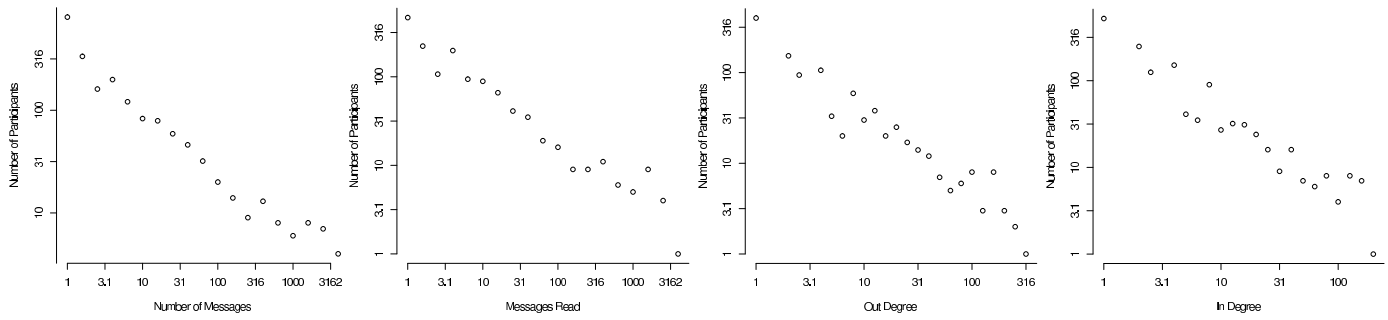


Figure 1: Note that all diagrams are log-log scale. Reading left to right: first, the distribution of people vs. number of messages they sent; next, vs. the number of reply messages they received. Note that a few people account for the bulk of the sending & reply activity. The next two indicate the structure of the social network. First, the out-degree in the social network; finally, vs. the in-degree in the social network. Out degree is an indication of status, as it indicates the number of different people who replied to the ego's messages. In-degree indicates the number of different people whose messages ego responded to. All distributions show power-law character. The degree distributions show small-world character of the email social network.

The first shows a histogram of message-sending behaviour. The vast majority of people send only one message, and there are some who send a great many. The second is the histogram of message replying behavior. The next two are based on the social network, where an individual s_a has a link to an individual s_b if s_b replied to a message from s_a. Higher out-degree for s_a is an indication of higher status, since more individuals have found messages from s_a of interest. Individuals whose messages attracted no replies were excluded from this graph. Out-degree also shows a scale-free, or power-law distribution, characteristic of small-world social networks [2, 9, 10, 15]. In-degree measures the number of different people to whom an individual has replied-to, and is an indication of the level of engagement of an individual in the mailing list and the breadth of his/her interests. This distribution also shows a small-world character.

Next, we examine (Figure 2) the relationship between the number of messages sent by an individual, and the number of distinct respondents who replied to that individual. For this graph, we only considered individuals who had actually received at least one response to their message. It can be seen that there is a strong relationship; in fact, we note a very high Spearman's rank correlation, around 0.97. It should be noted that these are not the same phenomena; the number messages one sends need not necessarily correlate with the number of different people that consider that message worth responding to. This may be due to community norms, *i.e.*, people only post relevant messages, and the community by and large responds to messages. It may also be due to a survival effect, whereby only people who receive replies from several people keep sending messages. We are currently using time-series regression analysis to examine the latter theory, that only people who receive replies to their messages continue to be active on the mailing list.

Finally we present (in Figure 3) a pruned email social network; the full network is too large to render in a useful fashion on non-interactive media. Each directional link in Figure 3 indicates a message count of at least 150. For example, the arrow from *Alexei Kosut* to *Ben Laurie* indicates that the latter replied to at least 150 messages from the former;

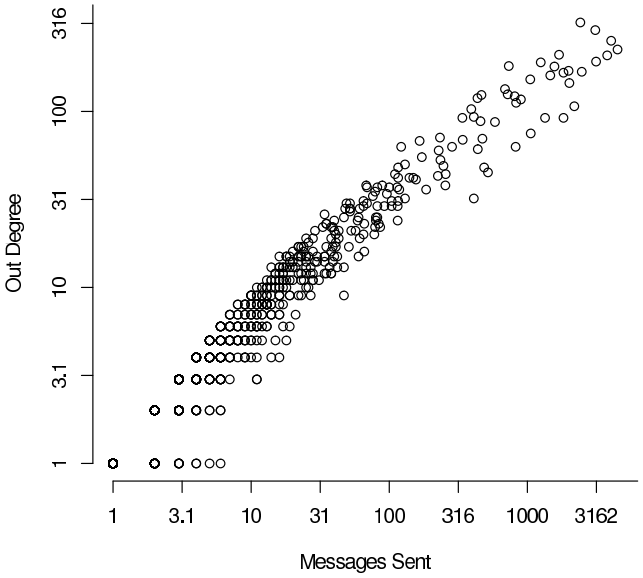


Figure 2: How out-degree (number of distinct respondents) grows with number of messages sent by ego, *n=1063*

this indicates that Laurie found a lot of Kosut's messages of interest. The reverse arrow indicates that this relationship was mutual. Unidirectional arrows, for example from *Slemko* to the *Rodent*, merely indicate that the former replied to less than 150 messages from the latter. Self links indicate that individuals sometimes replied to their own messages, sometimes after comments from others, sometimes to clarify their original message.

The high connectedness of certain individuals (*Gaudet, Laurie, Bloom, Jagielski, Rowe*) can be seen even in this pruned network; these individuals are in fact the most productive developers. Preliminary statistical data further supporting this is presented in the next section.

We conclude this section with some observations:

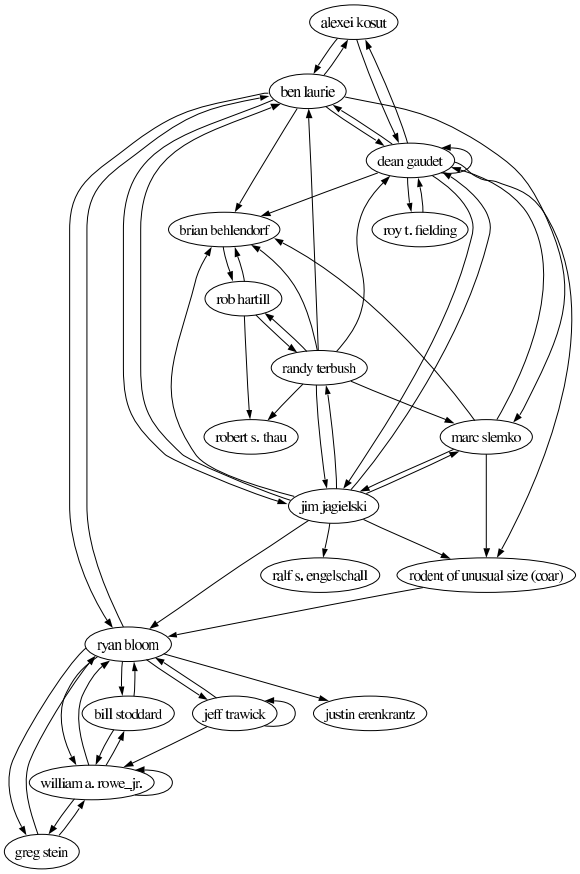


Figure 3: Pruned Social Network of Apache Emailers (Each link indicates at least 150 messages sent, or replied-to).

- *The number of messages sent by individuals, and the number of messages sent in reply to individuals, both follow a Pareto distribution;*
- *The social network of individuals on the email network, where an individual a has a link to an individual b if b replied to a message from a, shows a long-tailed degree distribution on both in- and out-degrees, characteristic of small-world networks.*
- *There is a strong relationship between the number of messages sent by an individual, and the number of* distinct *individuals who respond to that individual (also the out-degree in the social network). We are studying this phenomenon using time-series analysis.*

Next we turn to examine the relationship between email activity and development activity.

5. C&C ACTIVITY AND DEVELOPMENT ACTIVITY

In this section we discuss this question: *How does email activity relate to software development activity.* In order to study this question, we use data gathered from the cvs archives on how many changes (distinct commits) were made by each individual. In fact only 73 individuals have actually made commits to the versioned repository during the period beginning with 1999, (before which this repository was not used) until the present. There are two types of files, source and documents. We counted each separately, in order to study the relationship of source code and document activity with email activity.

5.1 Activity Correlation

There are large number of correspondents on the mailing list who do not have commit privileges, never make any changes to the project files. These individuals tend to be less active on the email list. In order to study the relationship between the effort spend on C&C activities, and development activities, we excluded individuals who have not made any changes to source code or documents from this study. By focusing on just those individuals who have made changes, we hope to get a clearer picture of the relationship of email activity with development activity.

Based on the data for just the 73 committers, we observe a Spearman's rank correlation of about 0.80 between the number of messages sent by an individual, and number of *source* changes they make. This clearly indicates that the more software development work an individual does, the more C&C activity the individual must undertake. There is a somewhat lower correlation, around 0.57, with number of *document* changes. We hypothesize that this is because source code activities require much more co-ordination effort than documentation effort, but further study, using time-series data is needed to determine this.

The total number of messages is only one aspect of a community's structure; the volume of messages sent by an individual (even if they receive replies) doesn't necessarily indicate the individual's position in the social network. Sociologists have invented several measures of an individual's position in a network, when viewed globally. We also study the relationship of some of these measures to the activity level of an individual.

5.2 Social Network Measures

We focus on 3 measures, *in-degree out-degree* and *betweenness*, which are indicators of the importance of an individual in a network. Out-degree and in-degree were discussed earlier; for this part, we normalize out-degree and in-degree by the total size of the network. For a node v in a graph g, betweenness BW is defined as follows:

$$BW(v) = \sum_{i,j,i\neq j,i\neq v,j\neq v} \frac{g_{ivj}}{g_{ij}}$$

where g_{ivj} is the number of shortest paths (geodesics)[3] in g, between i and j, that go through v; and g_{ij} is the total number of shortest paths from i to j.

High betweenness indicates that the person is a kind of broker, or gatekeeper in the social network; s/he plays a role in a great many interactions. Such people can have high status, and can also be bottlenecks. Actors who are high in betweenness centrality have the potential to control or disrupt communication or trust relationships between various end points. So we ask the question, *Are developers more likely to play the role of gatekeepers or brokers in the*

[3]Note that there may be more than one shortest path between two nodes if multiple paths are of the same length.

	changes	srcChanges	docChanges	outdegree	indegree	betweenness	mean	min	max
changes	1	0.789	0.932	0.520	0.474	0.553	912	0	16289
srcChanges	0.789	1	0.514	0.712	0.679	0.757	420	0	5741
docChanges	0.932	0.514	1	0.308	0.263	0.327	492	0	13420
outdegree	0.520	0.712	0.308	1	0.971	0.955	0.0080	0	0.0396
indegree	0.474	0.679	0.263	0.971	1	0.917	0.0067	0	0.0260
betweenness	0.553	0.757	0.327	0.955	0.917	1	0.0011	0	0.0965

Figure 4: Cross-correlation table, (using Spearman's rank correlation) showing relationship between the total number of changes, the changes to source, changes to documents, relative in-degree, relative out-degree, betweenness. Average, min and max are also shown. *n=73*

complete email social network? To answer this question, we computed the betweenness scores of developers ($n = 73$) and non-developers ($n=1123$) in the full email social network. The mean betweenness of developers is 0.0114, and the mean betweenness of non-developer is 0.000140. A simple T-test indicates a t-value of 5.07, which is highly significant. The other measures, out-degree and in-degree were also calculated. They also indicate that developers have a significantly higher status, as indicated in the table below.

	Developer	Non--Developer	T-value	Significance
Betweenness	0.0114	0.000140	5.07	$p < 0.001$
Out-degree	0.00666	0.000451	8.14	$p < 0.001$
In-degree	0.00794	0.000367	7.54	$p < 0.001$

So we can conclude that developers are higher in status than non-developers. Next, we consider just the population of developers, and study the indicators of status within this population.

5.3 Relative Status of Developers

Considering just the population of developers who have made changes to the source and documents ($n = 73$) we turn the reader's attention to Figure 4, which shows a table with the relevant descriptive statistics and correlation values. The top 3 rows (left 3 columns) are measures of activity: total changes, source code changes, and document changes. The bottom 3 rows (columns 4,5, and 6) are indicators of social status.

Considering just the 3 change variables, it can be seen that source changes are not as highly correlated with document changes, indicating that not all developers are engaged in both to the same degree. Thus, developer `nd` made 13420 document changes, and 2869 source changes, while developer `dougm` made 1322 source changes and 74 document changes. There are several others who were skewed in this way.

Turning now to the relative indicators of status, we can see that source changes shows the strongest rank correlation with the social network status indicators of normalized out- and in-degrees, and betweenness. In fact the correlation for betweenness is quite high, at 0.757. It should be noted that these are non-parametric correlation measures, and are thus more robustly indicative of a relationship. This indicates that even within the higher-status group of developers, the most active developers play the strongest role of communicators, brokers, and gatekeepers. It's also noteworthy that the correlation with document changes is much weaker, indicating that higher activity in source code is a stronger determinant of social status than activity in documents.

A later study of the developer mailing list and source code repository data for the Postgres[4] project showed that the social status measures had similar levels of correlation with source code changes [3]. The Postgres data, however, showed much higher correlations between document changes and social network measures than the Apache data. We plan to examine this statistic in future work.

We end this section with several preliminary conclusions:

- *The level of activity on the mailing list is strongly correlated with source code change activity, and to a lesser extent with document change activity.*
- *Social network measures such as in-degree, out-degree (normalized by the number of developers) and betweenness indicate that developers who actually commit changes, play much more significant roles in the email community than non-developers.*
- *Even within the select group of developers, there is a strong correlation between the abovementioned measures of social network importance and level of source code change activity.*

6. RELATED WORK

There has been considerable study of social behavior in on-line communities; we only survey work here in the OSS development context.

Social networks among developers have been studied from other perspectives. Xu *et al* [19] consider two developers socially related if they participate in the same project. Our view is to consider developers related if there is evidence of email communication; this is arguably a more direct evidence of a social link. Wagstrom, Herbsleb and Carley [18] gathered empirical social network data from several sources, including blogs, email lists and networking web sites, and built models of their social behavior on the network; these were then used to construct a simulation model of how users joined and left projects. Our goal is empirical rather than to run a simulation; we explicitly wish to study the relationship of email behavior and commit behavior in a single project.

Crowston & Howison [7] use co-occurrence of developrs on bug reports as indicators of a social link. They empirically demonstrate that the social networks of smaller projects are more central than those of larger projects, presumably larger

[4]http://www.postgresql.org

projects decentralize, to simplify C&C activities. This paper is more centered on the study of individual developers and how their email activity and social status changes with their commit activity.

Commit behaviour in versioned repositories has been used as indicator of social linkage. Lopez-Fernandez *et al* [12] consider two developers to be linked if they commited to the same module, and two modules to be linked if they were committed to by the same developer. The resulting social networks are similar in structure to ours. The work of De Souza *et al* [6] is similar, except that they study files instead of modules. This work also visualizes the changes in social position of developers within the social network over time, and that of modules in the module dependency network. Developers become more "central" in the social network over time. Turning to modules, they found that code ownership in some parts of the system was more stable than others. Finally, we note that these papers study collaboration networks, whereas our focus is more on communication networks; the relationship between the two is a subject of our current research.

7. CONCLUSION

We describe here our work on mining the email social network on the Apache HTTP server project. We had to face head-on the challenge of resolving multiple email aliases that were used by the same individuals; failing to do this would have seriously affected our ability to study the social network of developers. We have hand-inspected our alias resolution for errors; however, we acknowledge that our alias-resolution step is in need of further validation. Our goal is to do this by mailing a sample of participants get an idea of the accuracy of our alias resolution. Furthermore, a small number (less than 1.3%) of email headers could not be parsed; we are also working on resolving this. However, we believe, that a) there are likely to be only a few errors in the aliasing and b) that the preliminary results reported here are quite robust and unlikely to change significantly even as our data extraction improves.

Our analysis indicates that the email social network is a typical electronic community; a few members account for the bulk of the messages sent, and the bulk of the replies. The in-degree and the out-degree distribution of the social network exhibit typical long-tailed, small-world characteristics. We also note that there is a strong relationship between the number of messages sent, and the number of different people who respond to them; this merits further study.

Our preliminary data also indicates a strong relationship between the level of email activity and the level of activity in the source code, and a less strong relationship with document change activity. Our data also gives strong indications that developers play a much more significant social role among *all* the participants in the mailing list. Furthermore, the data also supports preliminary finding that the level of activity in the source code is a strong indicator of the social status of a developer (among other developers); the document activity is not as strong an indicator.

Our near-term goal is to study these effects in a time-series basis, to investigate if there are causal relationships between development activity and social status. We are also very interested in studying the relationship between the architecture of the system, and social network of the developers (which is also known as Conway's Law).

8. REFERENCES

[1] R. Agrawal, S. Rajagopalan, R. Srikant, and Y. Xu. Mining newsgroups using networks arising from social behavior. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 2003.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[3] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and M. Gertz. Mining email social networks in postgres. In *MSR '06: Proceedings of the International Workshop on Mining Software Repositories*, 2006.

[4] F. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley, 1995.

[5] S. Chapman. Sam's string metrics page. `www.dcs.shef.ac.uk/ sam/stringmetrics.html`.

[6] J. F. P. D. Cleidson de Souza. Seeking the source: Software source code as a social and technical artifact, 2005. http://opensource.mit.edu/papers/desouza.pdf.

[7] K. Crowston and J. Howison. The social structure of free and open source software development. `opensource.mit.edu/papers/crowstonhowison.pdf`, November 2004.

[8] B. J. Dempsey, D. Weiss, P. Jones, and J. Greenberg. Who is an open source software developer? *Communications of the ACM*, 45(2):67–72, February 2002.

[9] L. C. Freeman. Centrality in social networks I. Conceptual clarification. *Social Networks*, 1:215–239, 1979.

[10] M. Granovetter. The strength of weak ties. *American Journal of Sociology*, 78:1360–1380, 1973.

[11] K. Kuwabara. Linux: A bazaar at the edge of chaos. *First Monday*, 5(3), March 2000.

[12] L. Lopez, J. M. Gonzalez-Barahona, and G. Robles. Applying social network analysis to the information in cvs repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, 2004.

[13] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surveys*, 33(1):31–88, 2001.

[14] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[15] J. Nieminen. On centrality in a graph. *Scandinavian Journal of Psychology*, 15:322–336, 1974.

[16] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, Sebastopol, California, 1999.

[17] E. Ukkonen. Algorithms for approximate string matching. *Information & Control*, 64(1-3), 1985.

[18] P. A. Wagstrom, J. D. Herbsleb, and K. Carley. A social network approach to free/open source software simulation. In *Proceedings First International Conference on Open Source Systems*, pages 16–23, 2005.

[19] J. Xu, Y. Gao, S. Christley, and G. Madey. A topological analysis of the open source software development community. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*, 2005.

Geographic Location of Developers at SourceForge*

Gregorio Robles
grex@gsyc.escet.urjc.es

Jesus M. Gonzalez-Barahona
jgb@gsyc.escet.urjc.es

Grupo de Sistemas y Comunicaciones
Universidad Rey Juan Carlos
Mostoles, Spain

ABSTRACT

The development of libre (free/open source) software is usually performed by geographically distributed teams. Participation in most cases is voluntary, sometimes sporadic, and often not framed by a pre-defined management structure. This means that anybody can contribute, and in principle no national origin has advantages over others, except for the differences in availability and quality of Internet connections and language. However, differences in participation across regions do exist, although there are little studies about them. In this paper we present some data which can be the basis for some of those studies. We have taken the database of users registered at SourceForge, the largest libre software development web-based platform, and have inferred their geographical locations. For this, we have applied several techniques and heuristics on the available data (mainly e-mail addresses and time zones), which are presented and discussed in detail. The results show a snapshot of the regional distribution of SourceForge users, which may be a good proxy of the actual distribution of libre software developers. In addition, the methodology may be of interest for similar studies in other domains, when the available data is similar (as is the case of mailing lists related to software projects).

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Human Factors

*This work has been funded in part by the European Commission, under the CALIBRE CA, IST program, contract number 004337 and under the FLOSSWorld SA, IST program, contract number 015722. This work is based on the SourceForge database provided by University of Notre Dame, see details at http://www.nd.edu/ oss/Data/data.html.

MSR'06, May 22–23, 2006, Shanghai, China.

Keywords

Geographical location, mining software repositories, libre software, free software, open source software

1. INTRODUCTION

One of the most well known characteristics of libre (free, open source) software[1] is the worldwide distributed pool of developers that collaborate in tens of thousands of projects, using Internet-based tools for coordination. These projects are usually open to participation by anyone, from any corner of the globe, provided Internet access is granted; those with enough knowledge and skills can, in principle, join them. This openness, and the underlying informality, has resulted in an environment where participation is difficult to control, or even understand. One of the most significative examples of open issues in this respect is the geographical distribution of the aforementioned pool of developers. The answer to the question "where do developers live?" is not only interesting for academic reasons; it is also important from both strategic and economic points of view.

In this paper, we present a first approach to deal with this question by analyzing data about a huge sample of developers. We describe how we have mined the database of the largest libre software development supporting platform (SourceForge) looking for indicators to estimate the geographic location of the developers registered in it. Since the number of users of the SourceForge platform is well over one million, we can assume it is a reasonably good and representative proxy of the whole population of libre software developers (although for sure it presents some bias, as will be discussed later, for instance in terms of language knowledge).

The main goals of this paper are two: to show a methodology to estimate country of residence (as a simple quantifier of geographic location) using the indicators available in the SourceForge database, and to obtain a first estimation of the location of libre software developers.

With respect to the first goal, it is noteworthy to mention that SourceForge does not store specific information about the geographical location of developers, which therefore has to be inferred from other indicators, such as the domains in the e-mail address, or the time zone information developers introduce when registering at SourceForge. We believe that

[1]Through this paper we will use the term "libre software" to refer to any code that conforms either to the definition of "free software" (according to the Free Software Foundation) or "open source software" (according to the Open Source Initiative).

the methodology we have designed for this inference can be extended to deal with data from other sources, such as mailing lists.

With respect to the second goal, our estimation will be only as precise as SourceForge population is representative of the global libre software development population. We offer no proof of this representativeness, and therefore the results presented have to be considered with care. However, despite any bias the SourceForge population can have, it is the most global, diverse and (by far) largest community of libre software developers, which means that, even if the results were not extensible to the whole development community, they are interesting by themselves.

The structure of this paper is as follows. In the next section we present some other research efforts on the geographical distribution of libre software developers. Afterwards, while the third section contains a description of the data source we have used for the study, the forth one presents the methodology that we have designed to infer the nationalities. Next, the results of the application of the methodology to the SourceForge population is shown and briefly commented. Finally, conclusions and some ideas for further research are be offered in the last section.

2. RELATED RESEARCH

Among the several approaches to study the geographical location of libre software developers, we can identify two categories, according to the data acquisition process: those which collect specific data provided by certain libre software projects (such as the CREDITS files found with the source code, or information available on the web pages of the project), and those which obtain the data by surveying developers.

To our knowledge, the first study in this field [3] studied the meta-data that can be found in the Linux Software Map entries[2]. Among other fields, they contain the name and e-mail address of the main author. By studying the top-level domain[3] of the e-mail address, the country of residence could be partly inferred, although the presence of generic top-level domains[4] made it impossible to determine the location of many developers, especially those based in the United States. Hence, there is a bias, recognized by the authors, which recommend further research on this matter.

The Debian project was studied in 2001 [12], based on the country information that the Debian developers introduce in the Debian Developer Database. Since it also contains information about the admission date for each developer, an evolutionary analysis was performed, showing how Debian started primarily as an US-based project, turning later to an European majority. The presence of members of developing countries was minimal.

The CREDITS file of the Linux kernel, and the contact information of the GNOME project was also studied in 2001 [6]. Its most remarkable result is that the shift towards a more European-based development in both projects can be explained by economic theory, with the number (and distribution) of developers depending on the cost of opportunity. Some years later, a new study of the Linux CREDITS file [13] provided a more in-depth study of the geographical distribution of the kernel developers.

One of the first studies based on surveys was WIDI [12] (2001) which featured over 5,500 respondents. Results showed a majority of EU-based developers, although the self-selected nature of the participants introduced a bias which has to be taken into account. A later survey, FLOSS [4], was answered by about 2,500 self-selected developers over the Internet. Although it did not include the study of the geographical distribution, a surprisingly large quantity of European developers (in comparison with their American and Asian counterparts) participated. This was one of the reasons to perform similar survey with other *flavors*, such as FLOSS-US [2] (interestingly enough, Europeans where also predominant) and other Asian surveys.

Regarding SourceForge, it has been an inspiration for many research papers on libre software and software repositories in general. The most relevant to our work is maybe a statistical analysis of the projects hosted in SourceForge [5], which shows that it hosts many small to medium-sized projects, while larger ones (such as Linux, GNOME, KDE or Apache) tend to use their own development infrastructure. For our purposes, this is by no means a disadvantage, since many developers who contribute to large projects are also registered at SourceForge. We can, hence, consider SourceForge users population as the largest collection of libre software developers in the world.

3. DATA SOURCES

The data source analyzed in this work is the SourceForge database, as provided to research teams by the University of Notre Dame. The database is provided as a monthly dump under an special agreement[5]. Therefore, the data set we use is not public, but is available to the research community, which means that the results based on it are reproducible by other groups.

For our research, we use the *private* e-mail address and the time zone associated to every SourceForge user in the database. SourceForge uses the private e-mail address for verification purposes. It is private in the sense that it is not published in the site. The time zone can be specified by registered users, in which case it is used to localize the display of time when the user is logged in. Usually time zones contain the region and a city name (eg. Europe/Madrid), although there are other formats, such as abbreviations (eg. CET is Central European Time)[6]. The default choice, for users which have not selected a time zone, is GMT.

The SourceForge data is provided through a web-based tool. Queries on it are returned in a text file, with the database fields separated by semicolons. We have queried for the private e-mail and time zone fields, parsed the output, transformed it into and SQL dump, and fed a database with the data. After this process, we hold data for more than 1,180,000 registered users at SourceForge in November 2005.

[2]The Linux Software Map (LSM) is a database of software written or ported to Linux, `http://lsm.execpc.com/lsm/`.

[3]A top-level domain (TLD) is the last part of an Internet domain name; that is, the letters which follow the final 'dot' of any URL.

[4]A generic top-level domain (gTLD) is in theory used for a particular class of organizations (com for commercial organizations, edu for educational institutions, etc.). Those domains do not include geographic information.

[5]More information about this agreement can be obtained from http://www.nd.edu/ oss/Data/data.html

[6]For a complete list of time zones, visit: http://wwp.greenwichmeantime.com/info/timezone.htm.

This is by far the largest data set ever used to estimate the geographical distribution of libre software developers.

Another data set based on SourceForge, FLOSSMole [1], provides public information about its registered users, but only includes the information that can be retrieved from the public interface of the site. Therefore, it does not include the private e-mail address, which is basic for this study.

4. METHODOLOGY

The final goal of the methodology described in this section is to estimate, as accurately as possible, the geographical distribution of the users in the database, using the domain in their e-mail address and the time zone as the base for the analysis.

The inference is straightforward when the TLD (top-level domain) of the e-mail address corresponds with a country code (country-coded top-level domains, or ccTLD; table 1 displays a list with some ccTLDs and the country they are assigned to). This is for instance the case of one of the authors of this paper, who is registered at SourceForge with following e-mail address: *gsyc.escet.urjc.es* ('.es' is the ccTLD corresponding to Spain).

ccTLD	Country
de	Germany
es	Spain
fr	France
mx	Mexico
uk	United Kingdom
us	United States

Table 1: List of some country coded top-level domains (ccTLDs).

The same can be said for many time zone codes. Almost all countries have a time zone designated by region, or even an abbreviated one. For instance, the Europe/Madrid timezone is a good indicator about the developer being located in Spain. As in the case with ccTLDs, it is trivial to assign a time zone to a country (and therefore to a ccTLD). As a matter of example, table 2 displays some time zones and their corresponding ccTLDs.

Time zone	ccTLD
Europe/Berlin	de
US/Eastern	us
America/New_York	us
EST	us
Europe/Madrid	es
Europe/Moscow	ru
America/Sao_Paulo	br

Table 2: List of some time zones and country codes top-level domains (ccTLDs).

However, in many cases the identification of the country of origin is not that simple. The reason for this is basically because we have to face incomplete information.

4.1 Incomplete information

Unfortunately, from our total population of over 1,180,000 registered developers, more than 750,000 do not use a ccTLDs (67%) in their e-mail address. The use of generic top-level domains (gTLD), such as .com, .net, .org, .biz, .info, is widespread, and renders the identification of national origin more difficult.

With respect to time zones, around 425,000 have the default, GMT (38%). This is problematic, since in this specific case we cannot assume that the time zone was selected: maybe the user lives in any other time zone, but never set it, or maybe she lives in a GMT time zone (and therefore have correctly selected it). However, they should still be assigned to some national origin (since there are several countries with GMT time).

Fortunately, we can build upon the fact that we have both entries for all registered users, and one of them can be enough to have evidence about the country. This means that the 'problematic' records are only those that have a gTLD in the e-mail address and GMT time zone. There are about 280,000 users (25% of the total population) in this situation. Our aim in this section is to find ways to lower the percentage of users to which we cannot assign a country. Several methods will be used in this sense. We will start by inferring information from the second level gTLDs (SLDs).

Domain	Number
hotmail.com	63784
yahoo.com	40180
gmail.com	14191
aol.com	6275
gmx.net	4128
msn.com	3688
163.com	2013
ntlworld.com	1998
rr.com	1981
rediffmail.com	1881

Table 3: Top 10 domains in number of SourceForge users that have set GMT as their time zone (total: 66054 distinct domains).

Table 3 gives the top ten SLDs in number of developers with GMT as time zone. For the SLDs with many registered users, we can look for those who specified a time zone different from GMT, and, in a first approach, assume that users who specified GMT should have the same proportion of non-GMT time zones. In other words, we propose an algorithm to proportionally distribute those users with a GMT time zone among all other time zones found for a SLD (see figure 1 for a graphical display of this idea). So, the algorithm takes those SLDs with a gTLD and finds out the time zone that the corresponding users selected (from which we can infer the country). Then it assigns proportionally entries with GMT time zone to the given countries.

As a case of example, consider epo.org, a domain with 22 registered users. From these, 10 had set GMT as time zone, 8 had the Dutch Europe/Amsterdam, 2 had the German Europe/Berlin, 1 the Austrian Europe/Vienna and a last one the French Europe/Paris time zone.

The algorithm in this first approach would assume that the GMT entries have to be assigned proportionally to the other ones. This means that there are 10 entries to be split among the other countries. To make it proportionally, all non-GMT time zones are added up. The final estimation for each country is given by the sum of the original number of developers plus the proportional part of the GMT. Values

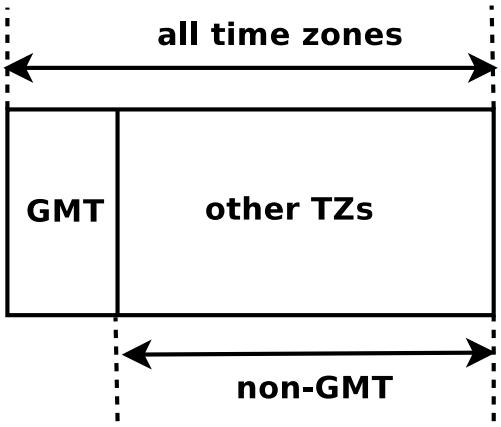


Figure 1: Redistribution algorithm graphically.

are not rounded at this point as this algorithm is going to be applied on all other gTLDs with GMT entries, so for the sake of accuracy rounding should occur at the end of the process. The following excerpt should clarify the algorithm:

```
GMT: 10 <---- to be distributed
nl:   8 -> 8 + 8/12 * 10 =  14.67 <- nl
de:   2 -> 2 + 2/12 * 10 =   3.67 <- de
at:   1 -> 1 + 1/12 * 10 =   1.83 <- at
fr:   1 -> 1 + 1/12 * 10 =   1.83 <- fr
                           -------
                  Total:    22
```

However, this algorithm presents problems for those countries which are actually in GMT, by underestimating their number of developers. Next subsection explains why, and shows a second approach which solves this problem. For the final estimation, this second approach will be used.

4.2 Countries in the GMT zone

The previous algorithm has a problem for those countries located in the the GMT zone, which in our data set are mainly the United Kingdom (uk), Ireland (ie) and Portugal (pt), since they are underrepresented. This is because we have assumed that those users who selected the GMT time zone did it 'by error' (never changing the default value). This is not always true, so we should find ways, to compensate this effect.

The basic idea for the subsequent reasoning is the assumption that those who live in the GMT time zone behave the same when filling out their data than the rest of the population. In other words, the 'error' rate of leaving the default time zone would be similar for all entries. Table 4 shows, for many European countries, the number of users with their 'own time zone' (time zone that corresponds to their respective ccTLD), and those that have selected GMT.

For instance, from those who have an Austrian (at) TLD, 3229 have chosen Europe/Vienna as their time zone, while 2840 left the default GMT. The last column shows the ratio between the own time zone and GMT. It is clear that these ratios are completely different for those countries that lay within the GMT time zone (with values below 0.3), when compared to the rest of European countries (with values in general between 1.10 and 1.90).

Country	own TZ	GMT	Ratio
at	3229	2840	1.14
be	4256	2701	1.58
ch	3813	2864	1.33
cz	2999	1708	1.76
de	36471	30857	1.18
dk	3779	2362	1.60
es	3930	2699	1.46
fi	3087	1187	2.60
fr	12150	8847	1.37
gr	1339	687	1.95
hu	2976	1957	1.52
it	12556	8917	1.41
lu	162	117	1.38
nl	9483	6027	1.57
no	2546	1620	1.57
pl	7607	4403	1.73
se	5817	3061	1.90
Total	116200	82854	1.40
ie	89	996	0.09
pt	632	2514	0.25
uk	2854	22108	0.13

Table 4: Time zone choice for some European countries.

If we take all European countries, the weighted mean of the ratio between the own time zone and GMT is 1.40. It is reasonable to assume that United Kingdom (uk), Ireland (ie) and Portugal (pt) should have a similar mean for that ratio. This assumption makes it possible to find a factor that can be multiplied to the entries corresponding to these countries in the GMT-assignation algorithm explained in the previous subsection. The equation for calculating this factor is:

$$Factor = \frac{GMT + ownTimezone}{1.71 * ownTimezone} \quad (1)$$

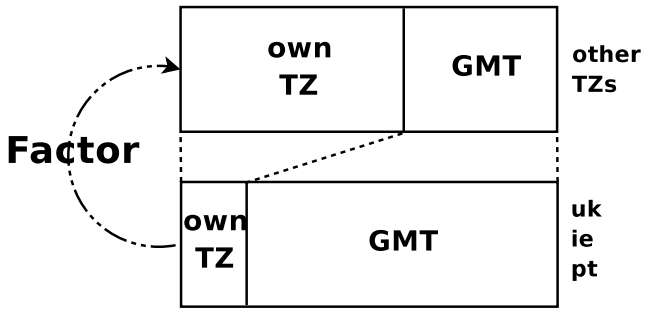


Figure 2: GMT factor calculation graphically.

How is that equation obtained? As shown in figure 2, Factor should ensure that the ratio of GMT to own time zone, for the GMT countries, is similar to that of non-GMT European countries. Therefore, we can define some conditions that must be met. As shown by equation 2, the number of entries before (given by the C, for current, subindex) and after (given by the F, for final, subindex), considering the factor, should remain constant.

$$GMT_F + ownTimezone_F = GMT_C + ownTimezone_C \quad (2)$$

A second condition is that the ratio between the final GMT and the final own time zone entries has to be 1.4, since this is the weighted mean of the ratio between the own time zone and GMT for the European countries (see equation 3):

$$1.40 * GMT_F = ownTimezone_F \quad (3)$$

A third condition introduces Factor (see equation 4), stating that the number of entries given for the final own time zone has to be the same found for the current one multiplied by the factor (i.e. the number of users remains constant).

$$ownTimezone_F = Factor * ownTimezone_C \quad (4)$$

Given these three conditions, GMT_F, $ownTimezone_F$ and $Factor$ are the unknown parameters. If we solve the systems of equations, we get equation 1, which shows how the factor is calculated.

The factors obtained for the GMT countries can be found in table 5. These values mean, for instance, that every *uk* entry for a domain should be weighted as 5.1 entries from other non-GMT countries when performing the redistribution algorithm presented above (and depicted in figure 1).

Country	Factor
uk	5.1
ie	7.1
pt	2.9

Table 5: Multiplying factor for GMT countries.

Finally, there is a small set of users (around 3% of the total sample) that have GMT as time zone, hold an e-mail address with a gTLD and do not share SLD with any other SourceForge user. For this set of developers we obtain the IP of the SLD by querying a DNS server. Using the geoIP library[7] we query for the geographical location of the host. The geoIP library contains a database that maps IPs to countries. This method is used, for instance, to assign a developer with the *hautpraxis.com* SLD to Germany (de).

4.3 Inferring geographical location

We have described several ways of obtaining the country of residence of our developer base. It can be done by looking at the TLD of the e-mail address, by transforming the time zone, by assigning the time zone proportionally from those given by the ones who share SLD, and if none of these are possible, by looking the geographical information of the SLD. All this means that we may have various information sources for a given developer and that information may be fragmented.

Figure 3 displays the four sets that we can find in our sample: ccTLD-other is the set of developers having an e-mail address with a ccTLD and a time zone different from GMT. ccTLD-other includes those with a ccTLD e-mail address and GMT as time zone (probably some of them will actually live in a GMT time zone, but many others just left the default). Those developers with a gTLD address and a non-GMT time zone are grouped in gTLD-other. Finally, gTLD-GMT contains those with a gTLD and GMT.

As we have seen in this paper, depending on the zone we can obtain the country of the developers by different means; sometimes by more than one for each set. For developers in ccTLD-other we could assign a country based on the ccTLD (method ccTLD) or from the time zone (method TZ). In the case of ccTLD-GMT, the assignation could be by studying the ccTLD (method ccTLD) or by redistributing the GMT time zone among the rest of time zones (method GMT-redist). The only possibility for those in the C set is to obtain the country from the time zone, while for D we can get it by redistributing the time zone among the SLDs. For those in D for which this is not possible (the ones who have set GMT and do not share a SLD with other SourceForge user that makes redistribution possible), the IP address of the SLD can be taken into account.

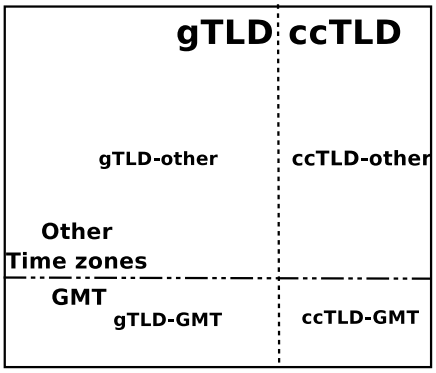


Figure 3: Set diagram with the different kinds of data.

Therefore, for ccTLD-other and ccTLD-GMT we have several choices. The yet unresolved question is to know which of them is better, since we do not know whether information provided by ccTLDs is more or less accurate than that obtained from the time zone.

5. RESULTS AND OBSERVATIONS

The results shown below have been obtained by assigning the ccTLD (so, we chose method ccTLD). We have also calculated results using method TZ (time zone instead of ccTLD) and GMT-redist (time zone distribution instead of ccTLD) and they are not significantly different.

Table 6 lists the top 50 countries by number of developers registered at SourceForge. These countries amount for 96.5% of the total identified registered users, while the top 20 countries include up to 83.9% of the total SourceForge population.

Although it is outside the scope of this paper, it would be interesting to find correlations between these data and some other per-country parameters, such as GDP or percentage of homes with Internet access. Just as a quick note, it is interesting to notice that this list is quite similar to the top countries by GDP, with some notable exceptions, of which the place of Japan (second by GDP) is probably the most surprising. Canada and Australia, on the other hand, are well above their ranking by GDP.

[7] `http://sourceforge.net/projects/geoip/`

Rank	Country	Developers
1.	United States	425620
2.	Germany	95800
3.	United Kingdom	60768
4.	Canada	49109
5.	France	44587
6.	China	36517
7.	Australia	31812
8.	Italy	30763
9.	Netherlands	29335
10.	Sweden	23867
11.	India	22113
12.	Brazil	21291
13.	Russian Federation	19012
14.	Spain	18905
15.	Japan	15081
16.	Poland	14697
17.	Belgium	13983
18.	Switzerland	12133
19.	Austria	10024
20.	Denmark	9952
21.	Singapore	9155
22.	Finland	9027
23.	Norway	8498
24.	Mexico	8185
25.	South Korea	7727
26.	Israel	6948
27.	Argentina	6695
28.	Hungary	6573
29.	Romania	6345
30.	Taiwan	6336
31.	Turkey	6099
32.	Czech Republic	6039
33.	European Union	5801
34.	South Africa	5706
35.	Portugal	4991
36.	New Zealand	4518
37.	Greece	4058
38.	Indonesia	3893
39.	Thailand	3746
40.	Bulgaria	3606
41.	Ukraine	3383
42.	Malaysia	3189
43.	Western Samoa	2856
44.	Ireland	2686
45.	Chile	2548
46.	Slovak Republic	2141
47.	Maldives	2067
48.	Colombia	2052
49.	Madagascar	1838
50.	Estonia	1758

Table 6: Results for the top 50 countries.

Also noteworthy is that we can find European Union in place 33 in the country list. This is because of the existence of the .eu domain and the CET time zone that could be assigned to a wide range of European countries (mostly part of the European Union). A way to address this problem could be using a redistribution algorithm for those entries, distributing proportionally among countries.

Surprising positions are achieved by Western Samoa (43), Maldives (47) and Madagascar (49). The reason for this is that some ccTLDs can be acquired without restrictions and have become *de facto* gTLDs. For instance, Western Samoa's top-level domain is ws, which has been sold as a shortcut for "website". Again, redistributing these entries in any of the ways already presented would make results more accurate. A good way of identifying these inflated domains would by correlating our results with total population, thus obtaining a per capita distribution. Per capita values that are too high will be a clear indicative.

Region	Developers
Africa	12560
Asia	127275
EU	401845
Europe	466792
North America	485679
Oceania	46422
South America	36330

Table 7: Results by regions.

Table 7 groups countries by regions. These figures are consistent with previous studies, maybe showing higher numbers for North America. In any case, it is clear that most of the developers come from Europe and North America (on an almost 50-50 ratio), followed by Asia with less than 10%. On the other hand, as the population is larger in Europe than in North America, this means that the *penetration* of the libre software development measured in SourceForge registered developers per capita is higher in North America than in Europe.

All of these results are of course not exact. We have worked with sources with rather different error margins, and we have used heuristics that are sound, but have for sure a certain error rate. To assess on the validity of the methodology for estimating the national origin, we should check (probably by contacting developers themselves) for a large fraction of SourceForge users. The results should then be compared with those of our study. However, the validations we have performed seem to indicate that the results are statistically sound, and that the figures shown are at least good estimators of the reality.

6. CONCLUSIONS AND FURTHER RESEARCH

In this paper we have described the process of extracting data about national origin from the SourceForge database, using mainly two parameters: e-mail addresses and time zones. We have also presented and discussed the results of applying this process to well over one million of registered users.

We have described the methodology with as much detail as possible, so that it can be completely understood and applied by third parties to this and other data sources. For instance, many methods described here can be used in other contexts, such as the study of contributions to the mailing lists archives of a project (provided there is access to the archives of those mailing lists).

Our methodology is not focused on identifying the geographical location of single developers (although in many cases that is done), but on finding the aggregate numbers of developers of a certain national origin. Therefore, we use in

many cases statistical relationships to infer the proportion of nationals of a certain country in a population of users with some characteristics. This is certainly a limitation of the proposed approach, specially if we were interested in (individual) developer identification methods as proposed in other works [10].

A future line of research could be to relate our findings with the activity of developers in the projects they are involved. This could be done by tracking developers in control versioning systems, mailing lists, forums, etc., and studying their activity by national origin. This could be an important issue, since previous research has shown that activity in libre software tends to be highly skewed towards a minority group responsible for the vast majority of the work performed. The authors of this work have started to analyze the CVS versioning system logs of all the SourceForge projects with the CVSAnalY tool [11], and the FLOSSMole project [1] has also information related to projects in the site. Both data sets could be used for this matter.

An interesting issue is how representative this study is of the whole population of libre software developers. SourceForge is not the only development platform: large libre software projects usually administrate their own infrastructure, and also many other SourceForge-like sites exist, in some cases linked to language or national communities. This means on one hand that we are not considering a lot of libre software which is being developed outside SourceForge (although many of the developers of that software are probably also users of this site), and on the other that the study could be skewed by ignoring some communities which are not represented in SourceForge, but in other facilities. Further studies should address this issue, and determine how good the SourceForge population is as a proxy of the developer population.

On a more socio-economic perspective, the findings presented in this paper could be related to other parameters characterizing the countries, looking for correlations which could explain the different quantities of developers, such as the GDP, the GDP per capita, Nielsen/Netratings, or other economic and technological parameters.

Especially interesting is also the issue of finding projects that are driven by local activity, i.e. projects whose contributors are from the same country, region or cultural environment. This could be a way of finding possible splits of the libre software community, and a first step towards identifying parameters leading to collaboration between developers. Cultural, language and other barriers should also be considered. In this sense, a recent change in the SourceForge platform has been the inclusion of a language field (although up to the moment less than 25% have specified a different language from the default 'English').

All of this could also be extended to a social network analysis, such as performed on libre software developers [8, 7, 9], but taking into account geographical information.

7. ACKNOWLEDGMENTS

We thank the SourceForge team, and Greg Madey from the University of Notre Dame, for providing access to the SourceForge data. Also, a big thank you goes to our colleagues from GSyC/LibreSoft for their help verifying the validity of the data.

8. REFERENCES

[1] M. Conklin, J. Howison, and K. Crowston. Collaboration using OSSmole: A repository of FLOSS data and analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 126-130, St. Louis, Missouri, USA, May 2005.

[2] P. A. David, A. Waterman, and S. Arora. FLOSS-US. The Free/Libre/Open Source Software Survey for 2003. *Technical report*, Stanford Institute for Economic and Policy Research, Stanford, USA, 2003.

[3] B. J. Dempsey, D. Weiss, P. Jones, and J. Greenberg. A quantitative profile of a community of Open Source Linux developers. *Technical report*, October 1999.

[4] R. A. Ghosh, R. Glott, B. Krieger, and G. Robles. Survey of developers (free/libre and open source software: Survey and study). *Technical report*, International Institute of Infonomics. University of Maastricht, The Netherlands, June 2002.

[5] K. Healy and A. Schussman. The ecology of open-source software development. *Technical report*, University of Arizona, USA, Jan. 2003.

[6] D. Lancashire. Code, culture and cash: The fading altruism of Open Source development. *First Monday*, 6(12), 2001.

[7] L. Lopez, J. M. Gonzalez-Barahona, and G. Robles. Applying social network analysis to the information in CVS repositories. In *Proc Intl Workshop on Mining Software Repositories*, pages 101-105, Edinburg, UK, 2004.

[8] G. Madey, V. Freeh, and R. Tynan. The open source development phenomenon: An analysis based on social network theory. In *Americas Conf on Information Systems*, pages 1806-1813, Dallas, TX, USA, 2002.

[9] M. Ohira, N. Ohsugi, T. Ohoka, and K.-I. Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In *Proceedings Intl Workshop on Mining Software Repositories*, St. Louis, Missouri, USA, May 2005.

[10] G. Robles and J. M. Gonzalez-Barahona. Developer identification methods for integrated data from various sources. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 106-110, St. Louis, Missouri, USA, May 2005.

[11] G. Robles, S. Koch, and J. M. Gonzalez-Barahona. Remote analysis and measurement of libre software systems by means of the CVSAnalY tool. In *Proc 2nd Workshop on Remote Analysis and Measurement of Software Systems*, pages 51-56, Edinburg, UK, 2004.

[12] G. Robles, H. Scheider, I. Tretkowski, and N. Weber. Who is doing it? A research on libre software developers. *Technical report*, Technische Universitaet Berlin, Berlin, Germany, Aug. 2001.

[13] I. Tuomi. Evolution of the Linux Credits file: Methodological challenges and reference data for Open Source research. *First Monday*, 9(6), 2004.

Textual Allusions to Artifacts in Software-related Repositories

Gina Venolia
Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA
http://research.microsoft.com/~ginav
gina.venolia@microsoft.com

ABSTRACT
Much of what is written about a software project is soon forgotten. Software repositories are full of valuable information about the project: Bug descriptions, check-in messages, email and newsgroup archives, specifications, design documents, product documentation, and product support logs contain a wealth of information that can potentially help software developers resolve crucial questions about the history, rationale, and future plans for source code. For a variety of reasons, developers rarely turn to these resources when trying to answer these questions. We are building a full-text search that encompasses multiple repositories. To effectively implement full-text search in the absence of hyperlinks we propose detecting *textual allusions* to software artifacts in natural-language prose. Allusions are shown to contribute a significant portion of the relationships represented in the graph.

Categories and Subject Descriptors
H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Retrieval models*; H.5.3 [**Information interfaces and presentation**]: Group and Organization Interfaces—*Computer-supported cooperative work*; K.6.3 [**Management of computing and information systems**]: Software Management—*Software development, Software maintenance*.

General Terms
Documentation, Experimentation, Human Factors.

Keywords
Software development, project memory, software artifacts, search.

MSR'06, May 22–23, 2006, Shanghai, China.

1. INTRODUCTION
In a series of surveys and interviews at Microsoft, my team learned that the most serious problem that software developers face is "understanding the rationale behind a piece of code" [4]. It's likely that this is a universal phenomenon, not limited to Microsoft. There are vast information repositories—bug descriptions, check-in messages, email and newsgroup archives, specifications, design documents, product documentation, product support logs, etc.—that have the potential to answer questions about rationale, but we found that developers rarely access them. Instead they examine the source code text and probe it in the debugger, and if those fail, they typically initiate a face-to-face conversation with the person they think will know the answer. This investigation process, while often successful, costs precious time and causes interruptions.

There are many good reasons for a developer to neglect the electronic repositories when trying to understand code. The developer does not know *a priori* whether a topic of interest is addressed in any repository. Fast full-text search is not implemented for all the repositories. Each repository has its own search and browse tools, and there's little consistency among them. It may be difficult to formulate a full-text query for the topic of interest, or to browse meaningfully for artifacts related to it. It may be difficult to assess whether an artifact (found by searching or browsing) is the last word on the topic or is hopelessly out-of-date. Given these barriers it's easy to understand why developers choose to neglect the electronic repositories.

To address these deficiencies developers need a good full-text search tool that spans the relevant repositories. Modern full-text search systems rely in part on link-analysis scoring algorithms, such as PageRank [5] and HITS [3], which estimate each artifact's importance based on analysis of the hyperlinks among the artifacts. Unfortunately hyperlinks are rare among software artifacts. This paper presents an approach to simulating hyperlinks by detecting *textual allusions* to software artifacts in the natural-language prose that is already present in many software artifacts.

1.1 Related work
The Hipikat system [1] provides artifact-based search for code-related artifacts. It combines structural relationships with relationships found by a measure of textual similarity.

Team Tracks [2], a recommender system for methods in source code, relies on implicit relationships discovered by aggregating

developers' navigation patterns in code to compute its recommendations.

2. REPRESENTING THE GRAPH

There are many types of software-related artifacts: source code files, classes, methods, bugs, check-ins, emails, specs, etc. While there are some common attributes across artifacts (e.g. a name, a brief description, and a date that the artifact first came into existence) each type may also have properties particular to it. Likewise there are many types of relationships: member-of, implements, mentions, addressed-to, etc. Together these requirements suggest that an appropriate representation may be a directed multi-graph where the nodes (representing the artifacts) and arcs (relationships) are typed.

Each node and arc in the index has an identifier that may be derived directly from its type and its identifying properties. In our present implementation each node has an *identifier*, which is a string composed of its type name and its identifying properties. For example the bug #12345 in the *AppBugs* database might be named "bug:appbugs:12345". The utility of this identifier will be covered in greater detail later in this section.

In the current implementation the nodes and arcs share a common abstract base class, `Entry`, properties for a unique identifier for the entry and the dates that the entry was created and deleted, which may be unspecified. Artifacts, or nodes in the graph, are represented by the abstract `Item` type, which derives from `Entry`. It extends `Entry` with properties for the human-readable name for the item, the string on which full-text search operates, and a URL or command for opening a viewer on the actual artifact (all are optional). Relationships, or arcs in the graph, are represented by the abstract `Link` type, which derives from `Entry`. It extends `Entry` with properties for the identifiers of the endpoint items, and an optional estimate of the confidence in the relationship.

The graph is represented in a persistent store called the *index*. The index is expected to be very large, so it is expected to be deployed as a shared resource. For these reasons the index is stored as a database using Microsoft SQL Server 2005. Stored procedures and web service APIs provide mechanisms for submitting, fetching, and deleting entries and for executing queries and retrieving results. Full-text search indexing is enabled for the `Item` name and search text columns.

When an entry is submitted to the index, the index first checks whether it already contains one with that identifier. If not then one is created with the specified property values; otherwise any newly-specified optional property values override the old ones. (The rationale for this behavior will be explained in section 3.4.)

This infrastructure provides a generic base on which a rich index of domain-specific information can be built.

3. PROVISIONING THE INDEX

An index is provisioned with artifacts and relationships from a collection of information repositories, e.g. bug databases, source code control system databases, email archives, etc. There are three distinct categories of provisioning—source schema, file structure, and textual allusions—discussed in the next three subsections.

3.1 Crawling the Source Schema

Each of the data sources has a unique programmatic interface, requiring custom software we call a *crawler*. The crawler uses the repository's interface to query for new information, creates instances of types derived from Item and Link to represent the new information, and then submits the instances to the index. The crawlers are run on a periodic basis, though in principle a crawler could be run when notified of new data by the data source.

For example consider the source-schema entries created by a crawler on a particular source code control system database. With this particular source code control system, check-ins are numbered sequentially. The crawler keeps a record of the last crawled check-in. When it runs, it queries the repository for the subsequent check-in numbers, and then iterates through each one, requesting detailed information about it. An instance of `CheckInItem` (i.e. the type derived from Item that represents a check-in) is created, along with a `DomainAccountItem` to represent the author, and an `AuthorLink` that connects the two items. Then, for each file revision in the check-in, the crawler created a `RevisionItem`, which is associated with the `CheckInItem` using a `ChangeLink`. A file revision is conceptually bound to a particular file path, so the crawler creates items to represent file itself and the directory hierarchy above it, associated with a chain of `ContainsLink` instances. The consecutive revisions are linked—a `RevisionItem` representing the previous revision is created and a `NextLink` relating it to the present `RevisionItem`.

Our current implementation has crawlers for the source code control system, the bug database, and email. In the future we expect to implement crawlers for Active Directory (the company-wide database that stores organization chart, email discussion list membership, and security group membership), a file system crawler, a website crawler, an RSS/Atom crawler for gathering weblogs, and perhaps others.

3.2 Cracking Structured Files

Structured files are an important source of artifacts and relationships in the index. Files occur many places in the repositories, e.g. as an attachment to an email or bug, or stored in a source code control system, file server, or web server. When a file is encountered it its type is used to look up a *cracker*, a piece of code that reads the file and produces items and links to represent its contents. For example a source code file contains useful structure, as does an XML file that controls a build process; on the other hand a Microsoft Word document has structure, but none that relates specifically to software-related artifacts.

The cracker "cracks open" the file and creates entries to represent its structure. The C# cracker runs a compiler front-end and walks the resulting abstract syntax tree to produce `ClassTypeItem` instances, `ImplementsLink` relationships to other `ClassTypeItem` instances, `FieldMemberItem` and `MethodMemberItem` instances and `ContainsLink` instances to associate them with the `ClassTypeItem`, etc. A build-file cracker would associate the items representing various source files with the item representing the binary output file.

Our current implementation has crackers for C/C++ files, which create shallow structure, C# files, which creates deeper structure, and any files for which an IFilter can be found (IFilter is public

Table 1: The number of items of each type, and their average number of incident arcs.

Type	Count	Avg. Degree
SCCS* file revision	2,688,714	2
SCCS file	878,736	3
SCCS check-in	379,913	8
SCCS folder	243,756	5
Identifier	190,177	4
Number	148,498	4
Bug	93,554	6
Bug revision	49,731	12
Local file path	17,436	3
Email message	12,203	47
HTTP URL	11,377	6
Email conversation	8,292	2
Domain account	8,067	70
Server file path	3,823	2
Email address	266	43
SCCS database	17	22,493
Bug database	4	23,388

* Source code control system

interface for components that convert files to plain-text streams; there are IFilters for dozens of file formats, including Microsoft Word, Excel, and PowerPoint, and Adobe PDF), which creates no structure but extracts a plain-text version of the file's contents, making it searchable and allowing it to be scanned for textual allusions. In the future we expect to implement a deeper cracker for C/C++, a cracker for Visual Studio project files, crackers for binary files, and perhaps others.

3.3 Scanning for Textual Allusions

Any text that is presumed to be natural-language prose that a crawler or cracker encounters is submitted to a battery of *scanners*, which scan the text for textual allusions to software-related artifacts and create entries to represent them. Each check-in crawled by the source code control system crawler has a check-in message which typically contains prose, as do comments in C++ source code, emails, word processing documents, and web pages. Each item type may contribute a scanner to the battery. We currently implement several scanners:

`EmailAddressItem`: Email addresses, e.g. "foo@bar.com", recognized with a simple regular expression.

`LocalLocationItem`: Local file paths, e.g. "C:\folder\foo.txt", recognized with a simple regular expression.

`UncLocationItem`: Universal Naming Convention file paths, e.g. "\\server\folder\foo.txt", recognized with a simple regular expression.

`IdentifierItem`: Uses a simple regular expression to find the kinds of names that are often used for software-related artifacts, e.g. "FooBar", "foo_bar", "foo123", etc..

`NumberItem`: Uses a simple regular expression to find strings of digits, e.g. "12345", because software-related artifacts are often numbered.

`HttpLocationItem`: While URLs are detected using a regular expression, redirection can cause a single page to have multiple URLs so the `HttpLocationItem` attempts to fetch any URL found by the regular expression, and uses the final URL to create the identifier.

`BugItem`: At Microsoft (and likely elsewhere), people use a wide variety of wording to reference bugs ("bug 12345", "resolves 12345", "duplicate of 12345", "fixes 12345, 23456, and 34567", etc.) the regular expression used by the `BugItem` scanner is much more complex than the others. At Microsoft there are hundreds of bug databases, with conflicting number spaces, so more work must be done to resolve the reference to a specific database. Most allusions to bugs rely on context to imply the particular database. The current system resolves the ambiguity by querying the index to find the bug database that's most strongly connected to the item that includes the scanned text. When a candidate bug database is detected, the `BugItem` scanner queries it for the specified bug number and then discards reference if the bug don't exist. (Note that we make no attempt to resolve vague allusions like, "that bug we worked on yesterday".)

For example, consider a hypothetical check-in message: "This fixes bug 12345, which was an off-by-one error causing an array scan to terminate before the end. It also caused that intermittent problem reported by foo@bar.com." The message contains a reference to a bug and a reference to an email address. When the `BugItem` scanner detects the bug reference, it creates an instance of `BugItem` and an instance of `MentionsLink` associating it with the present `CheckInItem`. The `NumberItem` scanner also detects a reference to the number 12345, and therefore instantiates a `NumberItem` instance and a `MentionsLink`. A similar process happens with the `EmailAddressItem` scanner. The other scanners are run but don't detect any allusions. Thus the knowledge casually coded into the check-in message is normalized into data structures.

In the future we expect to implement scanners for build numbers, knowledge base articles, domain account aliases, and method names. Method names might be approached with a combination of dictionary-based and regular expression techniques but both are problematic because, at least in current work practice, people often accidentally misspell identifiers and intentionally transform them into plurals (-*s*) and gerunds (-*ing*).

3.4 More about Provisioning

The crawlers, crackers, and scanners work in concert to provision the index with artifacts and relationships. They may be augmented by other techniques. Text similarity, used by the Hipikat system [1], could be applied to the index to create additional links, using the `Link` confidence property to represent the degree of similarity. Relationships between items discovered in navigation history, used by the Team Tracks system [2], could be turned into additional links (again using the confidence property), and indeed be extended beyond methods to include other artifacts such as bugs, emails, specs, URLs, etc. Simple rule-based approaches could be used to associate check-ins with contemporaneous bug actions by the same author. There may be other automated techniques for provisioning the index. They would work independently but the combined effect creates a richly-connected graph of software-related artifacts. In addition to automated techniques tools could allow the user to create items and relationships, such as annotations, and user-specified keywords that are automatically linked to the items that contain them.

Note that in several cases the crawlers, crackers, and scanners will submit items that may already be in the index. For example the bug database crawler may create a `BugItem` for bug 12345 and

Table 2: The number of links of each type.

Type	Count
Contains	5,578,988
Mentions	1,864,132
Next	1,590,770
Author	470,569
Recipient	49,118
Owner	43,683
Resolved-by	6,426
Closed-by	5,801
Reply	3,911

Table 3: The number of textual allusions by type of the item in which they were found and the type of item referred to.

		Mentioning Item Type				
		SCCS check-in	Email	Bug revision	Bug	**Total**
Mentioned Item Type	Identifier	221,510	197,301	298,173	224,436	941,420
	Number	309,403	209,630	89,139	66,431	674,603
	Bug	81,680	20,440	3,563	1,826	107,509
	HTTP URL	939	22,542	28,748	25,480	77,709
	Local file path	495	35,729	10,164	6,458	52,846
	Server file path	280	4,272	2,359	1,892	8,803
	Email address	21	773	241	207	1,242
	Total	614,328	490,687	432,387	326,730	1,864,132

the `BugItem` scanner may do the same. While the crawler has detailed information about the bug, and may thus populate the optional properties, the scanner knows only enough to create the item's identifier. To further complicate matters, either may encounter the bug first. The semantics of submission described in the section 2 allow the crawler and scanner (and any other mechanisms that provision the index) to operate independently.

4. RESULTS

We have built an index based on some of the data sources related to the development of the Microsoft Windows operating system. Activity between July 1st, 2005 and January 31st, 2006 has been gathered from eighteen source code control system databases one bug database. (For this analysis the contents of the source code control system file revisions were not gathered.) In addition, the index includes about twelve thousand emails dating from 2005 from several internal build-related email discussion lists.

The index includes 4,734,565 items and 9,613,398 links of various types (see Tables 1 and 2). (Note that each bug is composed of a series of *bug revisions*, each representing a specific action done to the bug: create, edit, resolve, close, etc.) While the average node degree (i.e. the average number of edges incident to the node) is 2.0, the distribution is highly skewed. The degree varies greatly by the node type, as shown in the *Avg. Degree* column of Table 1.

The links representing textual allusions are plentiful, comprising 19% of the links in the index. Table 3 categorizes the `MentionsLink` instances in the index by the type of item in which the allusion occurred and the type of item alluded to. (Note that text associated with a bug is counted twice, once for the bug revision and once for the bug itself; the *Bug revision* and *Bug* columns in Table 3 should be interpreted accordingly.)

5. DISCUSSION AND CONCLUSION

This effort combines the traditional representation of structured relationships with detection of textual allusions. These allusions contribute a substantial portion of the relationships represented in the index. Textual allusions are only one way to go beyond structured relationships. Text similarity, explored in the Hipikat project, and navigation paths, explored in the Team Tracks project, and other techniques, may complement the structured and allusive relationships.

The use of identifiers and the associated semantics of submitting items to the index combine to support decoupling of the various components contributing to the index. This is an important property if the system is to be allowed to grow organically, allowing new data sources to be added without much concern to the prior or subsequent additions.

While this initial work suggests that the approach is promising, there is a lot of work to do to evaluate whether it has benefits in real-world usage of tools built on the index. Tools that employ the index to deliver benefit to developers need to be fleshed out and evaluated in lab-based and field studies. Once deployed, their utility in helping developers understand the rationale behind code will become clearer.

If such a system is useful for software developers and their cohorts then it may be applicable to other knowledge work environments.

6. REFERENCES

[1] Davor Čubranić, Gail C. Murphy, Janice Singer, Kellogg S. Booth, "Hipikat: A Project Memory for Software Development," in *IEEE Transactions on Software Engineering* 31(6), IEEE Computer Society, pp. 446-465, June, 2005.

[2] Robert DeLine, Mary Czerwinski, and George Robertson, "Easing Program Comprehension by Sharing Navigation Data," in *Proc. VL/HCC'05*, IEEE Computer Society, pp. 233-240, 2005.

[3] Jon Kleinberg, "Authoritative Sources in a Hyperlinked Environment," in *J-ACM* 46(5), pp. 604-622, 1999.

[4] Thomas D. LaToza, Gina Venolia, Robert DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," to appear in *Proc. ICSE'06*, ACM Press, 2006.

[5] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Libraries Working Paper, 1998.

Enriching Revision History with Interactions

Chris Parnin, Carsten Görg*, Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{vector,goerg,spencer}@cc.gatech.edu

ABSTRACT

Revision history provides a rich source of information to improve the understanding of changes made to programs, but it yields only limited insight into how these changes occurred. We explore an additional source of information – program viewing and editing history – where all historical artifacts associated with the program are included. In particular, we suggest augmenting revision histories with the interaction history of programmers. Using this additional information source enables the development of several interesting applications including an influence-recommendation system and a task-mining system. We present some results from a case study in which interaction histories from professional programmers were obtained and analyzed.

Categories and Subject Descriptors: H.4[Information Systems Applications]:Information storage and retrieval.

General Terms: Measurement.

Keywords: Interaction history, revision history, data mining.

1. INTRODUCTION

During the process of developing software, programmers leave behind traces of their intentions, tasks, and missteps. Researchers have examined how to extract these traces for the purposes of acquiring additional insight into a program's history and the developers involved in the process. Artifacts created as byproducts during the development process offer a trove of insightful information – one popular artifact is the source code revision history.

Researchers have proposed how to use revision history in applications such as program evolution or defect detection. For example, Görg and Weißgerber [2] discovered incomplete refactorings that did not preserve semantics by analyzing revision histories. Alternatively, other researchers have examined how to use revision history to gain perspective into programming activities and developers. Sliwerski *et al.* [9] examined revision histories to find changes to a previous bug fix and analyzed which day of the week contributed to the most such incidents – Friday. Mierle *et al.* [5] attempted to find correlations between a student's grade and properties extracted from the student's revision history.

Revision history can be defined as a collection of revisions to a file committed by an author through a transaction or change request. Within each transaction, several pieces of metadata can be captured: date and time, change description, and change identifier or bug fix identifier. Furthermore, information about which methods have been changed between revisions can be derived.

Revision history transcribes various snapshots of source code; however, it has limited ability in explaining how the transition between revisions occurred. Information such as what methods were frequently referenced by programmers to perform a change or the order of edits is missing. Information detailing how a revision changed can be obtained through analyzing the interaction history. *Interaction history* records a user's interactions, captured by an application such as a viewer or editor, to understand more about the user and the program. All interactions are recorded into a stream of interaction events.

In attempting to understand a program's history, all possible artifacts should be used. Interaction history provides detailed information about a programmer's activities; however, the stream-like nature of interaction history makes segmenting into meaningful sessions problematic. A reasonable approach is to use revision history as the baseline for segmenting interaction history. With this segmentation, an interaction history session can augment the associated revisions. In the other direction, revision history provides more fine-grain details about the changes made to a program, while interaction history is primarily concerned with the locations of code involved with certain interactions. In the following table, the properties of revision and interaction history are contrasted.

properties	revision history	interaction history
metadata frequency	per transaction	per event
metadata type	time, log, ID	time, target, event type

In this paper, we propose that the interaction history of a programmer interacting with an interactive development environment (IDE) can be used in conjunction with revision history to enrich current approaches. We detail how to obtain the interaction history from an IDE and explain what properties to examine. Finally, we illustrate the use of interaction history in an influence-recommendation system and a task-mining system using interaction history obtained from a case study of professional programmers.

*The author was supported by a fellowship within the Postdoc-Program of the German Academic Exchange Service (DAAD).

MSR'06, May 22–23, 2006, Shanghai, China.

2. INTERACTION HISTORY

2.1 Background

An *interaction history* is a record of a user's interactions with an application for the purpose of providing insight into the data as well as execution of future interactions. Alternative terms for interaction history include *navigation history*, *user history*, *computational wear*, *edit wear*, *source code wear*. The first discussion of interaction history emerged from work on *edit wear/read wear* [3]. *Wear* is the concept of digital objects embedding the history of its interactions, much like the dog-eared pages in a book indicate favorite passages. As an example, a text document records how often a line was edited. The frequency of editing a line is then conveyed in a line-based visualization that is embedded in the scrollbar of the document.

Researchers have developed recommendation systems which analyze navigation history in order to recommend that locations of interest. In `FAN` [1], navigational history is analyzed to display a list of methods that are frequently accessed next after visiting the current method. In `NavTracks` [8], navigation loops are recovered from recent navigation paths and the files related to the current method are displayed. Both `FAN` and `MYLAR` [1, 4] have used a degree of interest model based on edit and navigation frequency to indicate the 'hot spots' in source code. Finally, Schneider *et al.* [7] have used interaction history to support awareness of activities among a team.

Previous research with interaction history has focused on understanding navigation patterns and deriving simple frequency statistics. We propose a set of abstractions over interactions that allow more interesting analyses to be performed.

2.2 Abstracting Interactions

Programmers interact with source code through an IDE and revision control system. Abstractions of these interactions allow us to reason about the semantic implication of different interactions with source code entities (in this paper we assume methods).
The categories of interactions with an IDE are the following:

navigation: A command used to go to a specific location in a file such as a to method.

click: A mouse selection of a method.

edit: A change in a line of code.

inspect: An examination revealing the metadata associated with a method such as a comment or type information. This is typically accomplished by hovering the mouse over the method.

query: A search for the locations of a method.

shelve: A text editing operation on code, such as a copy or paste.

Interaction history is represented as a stream of interaction events, where each event is a tuple of method, interaction type, and timestamp. An example is as follows:

$$[(A, click, 1), (A, edit, 2), (A, copy, 3), (B, paste, 4), (C, nav, 5)]$$

When using a revision control system, programmers also interact with source code. A list of interactions would include:

revision: A set of changes made to a file.

tag: A set of metadata such as comments, bug numbers, and change packages associated with a revision.

2.3 A Visual Studio Plug-in

We built a Visual Studio plug-in called `InteractionHistoryDB` that records the interaction history of programmers using `Visual Studio`. The plug-in registers interactions exposed through the `Visual Studio` add-in interface and logged the active method targeted by the interaction.

We identified six interaction types, however, we only recorded *click*, *navigation*, *shelve*, and *edit* interactions in our experiments. Click events were recorded using a mouse-message hook. Navigation events included commands such as *goto definition*, change active tab, select a class or file, navigate from a *find-in-files* result. Edit actions were recorded by listening to events raised when a line of code was changed. The edit event is not raised until after the user changes focus from the line being edited.

2.4 Case Study

Ten employees from a defense contractor volunteered to participate in a trial use. The projects the employees worked on were written in C++ and C# and varied in size from 50k to 200k lines of code. Some projects were over ten years old while others were in new development.

3. METADATA ANALYSIS

In analyzing program history, the questions asked about the history are generally motivated by two different perspectives:

program-oriented: Analysis focuses on retrieving code that satisfies a given query.

developer-oriented: Analysis focuses on understanding properties associated with a developer.

Detecting bad smells in code is an example of program-oriented analysis, while *determining what day of the week do the most navigations occur* is an example of developer-oriented analysis.

Regardless of perspective, the approach in analyzing the metadata of interaction history is much like revision history; however, different types of interactions are examined, and the granularity of the events are more fine. In general, revision history is a record of *what* happened, while interaction history is a record of *how* changes happen.

Localized frequency is the analysis of how events occur over a period of time. This analysis can be used in forming models of programmer interest and understanding the structure of programming activities. Later in this section, we provide examples of how interactions from interaction history can be used to enrich analysis.

3.1 Localized Frequency

The period of time or frequency that a programmer interacts with a method can be used to indicate interest. When users navigate source code, they often "thumb" through the code to locate the next method of interest. This can produce some interactions which are not desirable for analysis.

With localized frequency, a model of interest used in queries for analysis or filtering interaction data can be derived to mitigate this problem. We define this model as *intensity*.

Intensity. The intensity of an interaction with a method is *the number of prior consecutive interactions with the same method during the period of interest*.

For the event stream "AAABBC", the respective intensity of each interaction event would be 0,1,2,0,1,0.

Although a programmer may interact with many methods in the course of a session, only few exhibit high intensity. One possible interpretation of intensity is that method groups with high intensity are the targeted methods of interest to a programmer. In the *valleys* between two peaks of high intensity are smaller peaks of medium and low intensity activity. These lower intensity activities can result from the need to correct compile errors, update references, and search for the next item of interest.

Intensity takes a simplified, discrete view of localized frequency; in reality, a programmer may need to briefly transition away from a method and then return. This requires a continuous evaluation of localized frequency that we call *momentum*.

Momentum. The momentum at time t_n of an interaction event is:

$$momentum(t_n) = intensity(t_0) * e^{-rt_n}$$

where r is the *discount rate* which regulates the speed of exponential decay, t_0 is the time the event commenced, and t_n is n steps after t_0.

Instead of having a value of zero after a transition, momentum exponentially decays the intensity. An interaction event having an intensity of 20, with a discount rate of 0.1 would have a momentum of 7.4 after ten steps and a momentum of 1.0 after 30 steps. There is a small twist: a method that is decaying can be reinvigorated when revisiting the method. In this case, the remaining momentum is accumulated with the newer intensity, and t_0 is reset to be the new time.

Momentum gives a better measure of which methods are active during a window of time; however, its continuous nature can make it more difficult to apply in some situations.

3.2 Example Queries

What term is most commonly searched? Analyzing the distribution of search terms may reveal items difficult to locate, items not immediately understood by the programmer, or items relevant to the current task. Filtering out events without query interaction types results in the following example interaction event stream:

$$[(\text{“}display\text{”}, query, 1), (\text{“}screen\text{”}, query, 56), (\text{“}display\text{”}, query, 78)]$$

In this stream, 66% of the queries were for “display” and 33% for “screen”.

What is the ratio of transitions to edits? Understanding the relationship a programmer’s edits and transitions between methods in a project inspires several applications: (1) it serves as a baseline for comparing tools in experimental studies, (2) the rate of change of the navigation/edit ratio can be used to determine when a programmer is searching, and (3) it can assist in classifying tasks applied to the same project.

Consider the following event stream as an example:

$$[(A, click, 1), (A, edit, 2), (A, click, 3), (A, edit, 4), (B, click, 5)]$$

To avoid a heavily edited method from imposing too much bias on the ratio, navigations within methods (in our example (A,click,3)) are first removed, and then from the resulting stream the consecutive edits within a method (in our example (A,edit,2) and (A,edit,4)) are considered as one edit. After this preprocessing, the stream appears as follows:

$$[(A, click, 1), (A, edit, 2), (B, click, 5)]$$

The navigation/edit ratio for this stream is 2.0.

What time of day has the highest activity? Activity analysis can assist in studies of work patterns or in giving practical guidelines of when to schedule meetings.

To calculate this measure, the interaction event stream must first be segmented into different sessions corresponding to each hour of the day. Then, the length of each session belonging to the same hour is accumulated and averaged.

In data from our case study, 2-3pm was the period of highest activity for almost all programmers.

4. APPLICATIONS

In this section, we present two systems focusing on program-oriented analysis of interaction history.

4.1 An Influence-Recommendation System

Programmers frequently copy and paste code during the development process. The purpose of copying code varies: sometimes the programmer is avoiding the need to retype a variable name, or the copied code is being used as a template for a new variation.

Method A is *influenced by* method B if B contributed to the implementation of A through the importation of code. The influence set of A can be calculated from an interaction history by finding the origin of pasted code.

$$Influence(A) = \{B \mid \text{copy from B and paste to A}\}$$

In the following interaction event stream:

$$[(A, copy, 1), (B, paste, 3), (C, paste, 5), (C, copy, 6), (D, paste, 8)]$$

the influence sets are the following:

method	A	B	C	D
influence set	$\emptyset$	$\{A\}$	$\{A\}$	$\{C\}$

strength: The number of lines copied.

complexity: The cyclomatic complexity of code copied.

support: The number of occasions the method was influenced by the same method.

Finally, with the influence sets available for each method, an influence-recommendation system can be built. When a programmer is interacting with a method, the influence set can be used to recommend to the programmer items of interest in a contextual list displayed in an IDE. In preliminary analysis of our case study data, influence sets found (1) methods complementary to a process such as starting and stopping a server, (2) code duplication, and (3) methods serving as sources of examples (*e.g.* how to invoke a socket function call).

4.2 A Task-Mining System

Consider these situations:

1. During development of a major product release, the project manager anticipates the need to add support for a new system. The program developers interleave their normal programming tasks with integrating support for the new system over several months of development. Unfortunately, the project manager’s gamble does not pay off; the customer decides not to use the new system. As a result, the programmers need to identify and remove the changes that were made to support the new system.

2. A program developer needs to perform an update to an established project. The source code contains millions of lines, but the update should only involve a relatively small subset. The programmer would like to explore the program as efficiently as possible in order to make the changes, but is too unfamiliar with the relevant parts of the source code to be sure how best to proceed.

The tasks facing our two users are different; however, in both cases, we have a user who is interacting with a large set of highly-structured data and attempting to solve a specific task. Further, their tasks require connecting various parts of the data (*e.g.* different files or methods) that are relevant in solving their task. Discovering or recalling these connections is costly; the connections may not be readily apparent from the structure, or only a few relevant pieces of information are needed among a large selection of data.

In the first scenario, identification of different tasks performed with the source code can ease the search for methods related to implementing the new system. To identify the distinct tasks performed by the programmers, clustering of the interaction sessions can be performed. Further, labels from revision history supplement identification of different sessions.

In the second scenario, providing recommendations for related interactions enables the programmer to focus on understanding just the relevant methods. Recommendations can be queried based on the last few interactions of the programmer with the IDE.

To mine these tasks from interaction history, the data must first be segmented into different sessions and then represented in a vector space model (VSM). The straightforward approach is to use revision history to group interaction events related to the same groups of transactions in the same session. Each session is then represented as a vector by counting the occurrences of a *method* and its *interaction type*.

Distance metrics measure the similarity of two sessions. Two common metrics are the euclidean distance and the cosine distance. The euclidean distance uses the L^2 norm or dot product of two vectors. The cosine distance measures the angle between the vectors normalized by their magnitude. In both metrics, a higher value indicates a higher similarity.

In the following example, three sessions are encoded as vectors and stored in a matrix. In the first session, "payroll" was edited five times, "print" was navigated to three times, "report" was navigated to two times. In the first session, the methods "employee", "sales", "dbquery" were not interacted with.

$$\begin{pmatrix} payroll.edit:5 & employee.nav:3 & dbquery.edit:6 \\ print.nav:3 & payroll.edit:5 & employee.nav:3 \\ report.nav:2 & sales.edit:1 & sales.nav:2 \end{pmatrix}$$

After the sessions are represented in the VSM, several refinements can be made. Methods that are frequently or rarely interacted with can be weighted using measures such as tf-idf [6].

To identify distinct tasks, a standard clustering algorithm such as k-means can be performed. Reconstruction of the vector space with transformations such as independent component analysis may be necessary in order to more easily discriminate different tasks.

Using implicit query, the IDE can generate recommendations based on the closest session or centroid. If the programmer visited "employee" and edited "payroll" then an implicit query could be constructed as follows:

$$\begin{pmatrix} employee.nav:1 \\ payroll.edit:1 \end{pmatrix}$$

producing the following table of relevant interactions:

	v_1	v_2	v_3
dot-product	5	8	3
cosine	0.57	0.96	0.31

In this case, v_2 is returned as the most relevant interaction vector. An IDE, can now suggest editing *sales* through a degree-of-interest visualization or in a task pane.

5. CONCLUSION

In this position paper, we show that by incorporating interaction history with traditional approaches used with revision history, further insight can be gained. In program-oriented analysis, interaction history enriches analysis by introducing data previously unattainable and provides more options for making decisions in filtering data. In developer-oriented analysis, more data is available for understanding how the programmer performed tasks.

We conclude that interaction history: (1) can be easily obtained, (2) contains a rich source of interesting data, (3) offers powerful applications in recommendation systems, and (4) complements revision history as a source of insights into programmer behavior.

6. REFERENCES

[1] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 183–192, New York, NY, USA, 2005. ACM Press.

[2] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR'05: Proceedings of the International Workshop on Mining Software Repositories*, New York, NY, USA, 2005. ACM Press.

[3] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 3–9, New York, NY, USA, 1992. ACM Press.

[4] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[5] K. Mierle, K. Laven, S. Roweis, and G. Wilson. Mining student cvs repositories for performance indicators. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[6] G. Salton and C. Buckley. Term weighting approaches in automatic text retrieval. Technical report, Cornell University, Ithaca, NY, USA, 1987.

[7] K. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *MSR '04: Proceedings of the 2004 international workshop on Mining software repositories*, pages 106–110, 2004.

[8] J. Singer, R. Elves, and M.-A. D. Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM 2005: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 325–334. IEEE Computer Society, 2005.

[9] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, New York, NY, USA, 2005. ACM Press.

Using Evolutionary Annotations from Change Logs to Enhance Program Comprehension

Daniel M German
Dept. of Computer Science
University of Victoria
dmg@uvic.ca

Peter C. Rigby
Dept. of Computer Science
University of Victoria
pcr@uvic.ca

Margaret-Anne Storey
Dept. of Computer Science
University of Victoria
mstorey@uvic.ca

ABSTRACT

Evolutionary annotations are descriptions of how source code evolves over time. Typical source comments, given their static nature, are usually inadequate for describing how a program has evolved over time; instead, source code comments are typically a description of what a program currently does. We propose the use of evolutionary annotations as a way of describing the rationale behind changes applied to a given program (for example "These lines were added to ..."). Evolutionary annotations can assist a software developer in the understanding of how a given portion of source code works by showing him how the source has evolved into its current form.

In this paper we describe a method to automatically create evolutionary annotations from change logs, defect tracking systems and mailing lists. We describe the design of a prototype for Eclipse that can filter and present these annotations alongside their corresponding source code and in workbench views. We use Apache as a test case to demonstrate the feasibility of this approach.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Documentation

Keywords

Software evolution, mining software repositories, evolutionary annotations, version control

1. INTRODUCTION

It is undeniable that the most desirable property of source code is that it performs that task it is intended for, and it is also a well-known problem that source code lacks proper software documentation (documentation that describes how a system is implemented including source code comments). While some developers argue that source code should be self explanatory, it is widely acknowledged that software documentation is an important source of information that assists developers during comprehension and maintenance [6]. The primary goal of software documentation is to describe what the system does *currently*, and how it is implemented *currently* .

As a software project evolves, a wealth of information is created (some automatically, some manually) that describes how a software system is evolving. We have previously demonstrated that historical records can be used to successfully reconstruct how a software system evolves [2]. Our hypothesis is that this information, combined with software documentation, can improve comprehension and maintenance too.

In this paper, we propose the concept of evolutionary annotations, documentation that describes how a software system is evolving, and a method for their automatic retrieval from historical software development records such as version control logs, mailing list discussions, defect tracking databases. We also describe some of the challenges of extracting this information and correlating it with the source code. We conclude with a description of a prototype for the Eclipse Java development environment [1] that displays evolutionary annotations related to the source code.

2. EVOLUTIONARY ANNOTATIONS

System documentation evolves, whether it is internal or external to the code. Ideally documentation should evolve in tandem with the source code. One of the important goals behind software documentation is to explain what the current version of the source code does. What it lacks typically is a record of how the source code evolves, and the decisions that led to its current form.

We define evolutionary annotations (EAs) as documentation that explains the change or evolution of a software system rather than its current role. EAs reside in various places:

- Change logs. Files that describe what is changed at any given point (sometimes this information is embedded as an ever-growing comment at the top of the file). It typically contains a timestamp and a brief description of the change.
- Configuration management. It can explain the entire provenance of a change: who requested it, why, who implemented, who approved it, etc.
- Version control system. It keeps track of how source code changes: who performed the change, what was changed and when. Its logs usually contain an explanation of the rationale for the change. Sometimes version control systems are part

[1]See `www.eclipse.org`

MSR'06, May 22–23, 2006, Shanghai, China.

of a configuration management system, but more frequently version control systems exist on their own.

- Defect tracking system. It might explain the bug or feature that the change fixed or implemented: who found it, who fixed, test cases, explanation of the fix, etc.
- Mailing lists and newsgroups messages. Email and newsgroups postings that discuss or describe how the system evolves. The scope of these messages might vary, as some might describe how the entire system is evolving, while others might be very specific to a given change.
- Records of code reviews. The rationale behind a change that results from a code review might be very useful in understanding why a given change was performed in a certain manner.
- TODO tasks (such as the ones described in [7]). Following their evolution could provide valuable information about how changes are requested and who completes them.
- Comments in the source code itself. Some comments are indeed a description of how the source has changed.

In some instances a change in documentation might trigger the creation of an EA. For example the removal of a TODO task which might, but not necessarily, correspond to the completion of a task; or the removal of a source code comment that might imply a major change in the source code around it (for instance, an old algorithm is no longer used and the source code comment is no longer applicable).

If documentation that explains what a program does is seen as "vertical" (contained within a single file), evolutionary annotations are then "horizontal" (span at least two versions of a file), i.e. they are orthogonal and they complement each other. Figure 1 demonstrates how evolutionary annotations explain how source code files change between versions.

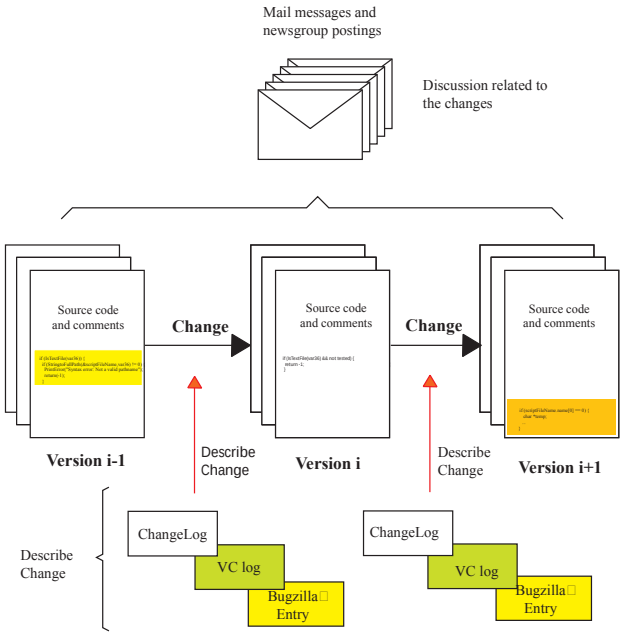


Figure 1: Evolutionary annotations are "horizontal" while source code comments are vertical.

Evolutionary annotations, like any other type of historical information, will grow as time passes by. It is necessary to present them to the reader in such a way that they are meaningful. EAs need to be filtered and ranked in such a way that only the most meaningful are presented to the reader. The definition of "most meaningful" might also change depending on the task at hand. A simple method to filter EAs can be based on their attributes. EAs can have the following inherent attributes:

- Type. This classification corresponds to the source of the annotation: emails, ChangeLogs, defect tracking, source code comments, etc.
- Scope. Evolutionary annotations, like the source code itself, have scope. Some describe changes at the global level (e.g. one might explain why the architecture of a system changed) while others might be at the line level ("this *if* statement was added to fix bug number..."). In some cases, the scope of a EA might not be related to a scope in a programming language sense (i.e. a EA that relates a global variable with some lines of code in several functions); thus, the annotations of some EA might not have an equivalent source code scope.
- Timestamp. When was the annotation created.
- Author. Who created it.
- Version/Revision. Indicates which version of the source code the EA correspond to.
- Community/project defined. Evolutionary annotations can be further refined with the use of keywords or other special fields that describe them. They can be enhanced according to the needs of the project. For example, developers can rank them according to importance, or can add keywords that are meaningful to their application domain. Ideally these refinements will be defined and created in such a manner that they improve query and visualization mechanisms.
- Type of change. EAs should also be labeled according to the type of source code they document (such as defect fixes, structural changes, new functionality, refactoring etc).
- Other EAs. Further annotations can be attached to a given EA that might provide a more in-depth explanation of the change.

Some of these attributes–such as author, timestamp and type–are easy to compute. Others, however, are not that simple. It will be difficult, for example, to automatically determine the scope of a comment. One potential solution is to allow the developers to further annotate EAs with their scope, either during their creation, or when they are being explored.

Automatically ranking annotations according to their relevance for the task the reader needs to complete, is an open problem and we expect to conduct further research in this area. The notion of decay should also be supported. As the code evolves, some, but not all EAs will lose their importance. We believe it will be difficult to automatically determine the rate at which a given EA should decay. Any system that presents EAs to the user should be able to take advantage of these properties in such a manner that the user can order them and filter them to meet their information needs.

3. EXTRACTING AND CROSS-REFERENCING EVOLUTIONARY ANNOTATIONS

One of the main challenges in the creation and maintenance of any type of documentation is to convince developers to create it, and then to keep it up to date.

EAs have an advantage over traditional documentation in that they do not need require user interaction to be kept up-to-date. Like any historical record, they need to be created when such an event happens; after that they might never change again (except, as we mentioned above, by adding extra annotations to them). Therefore the challenge is to get developers to create them in the first place.

Some evolutionary annotations are created automatically as a result of a change. For instance, if several lines of code in two different functions are committed at the same time, an evolutionary annotation is created that links them to each other, and with their author. This annotation can be further enhanced by its author by providing an expressive version control log description. Many evolutionary annotations are, therefore, created by developers as a result of their daily activities.

Since open source communities generally interact in an asynchronous, distributed manner, records of all changes and discussions are captured and stored. These records serve as the raw data for the creation of evolutionary annotations (EAs are the links between these records). In our research into the evolution of open source projects we have found that developers of mature open source projects value these records and ensure, often through policy, that these records be maintained; they form what Cubranic calls the "community memory" [1] of the project. We have observed that:

- ChangeLogs are usually updated with a change. The Free Software Foundation requires all its projects to have a ChangeLog file. In those projects that have them, we have discovered that they are almost always updated [3].
- Version Control logs tend to have large, meaningful explanations. In the project Evolution, the average size of a log is 306 bytes, in Apache 1.3 it is 160 bytes, and in Postgresql it is 160 bytes, to cite just a few.
- Email is seen as an important source of discussion about the way software evolves. For example, the Apache HTTPd server conducts code reviews on many of its patches (some pre-commit and some after they have been committed). The discussion is often lively with reviewers providing detailed explanations as to why a certain approach is good or bad [5]. In contrast, version control logs and comments are shorter, usually omitting discussion of less satisfactory solutions. Having a link to this discussion will likely save the maintainer many hours in code comprehension and avoids time wasted to re-implementing known poor solutions. In the case of Apache code review information is archived, but (to our knowledge) it has not been cross-referenced or linked to the source code. Without this link it is difficult to to know, for a particular function/file/line of code, what discussion has occurred.
- Defect tracking databases, such as Bugzilla, are frequently found in large open source projects. They provide a valuable source of information regarding defects (and their fixes) and feature request.

Some data sources that have a well defined format, such as version control logs and ChangeLogs, are easy to correlate to lines of affected code. Correlating Bugzilla and source code is more difficult. It usually involves textual analysis of the description of the version control log. For example in [3], we describe regular expressions that were useful in the extraction of Bugzilla numbers from CVS commit logs. Correlating email messages is even more difficult. For Apache, we have been successful in creating automated and manual heuristics that help in the correlation of messages discussing code reviews [5]. For example, code reviews often involve *diffs* that contain the version in the repository against which the diff was made. However, general email discussions are much more difficult to correlate. Problems include determining the context of the discussion, reconstructing message threads, and resolving names to email addresses. We expect that different projects will require variants of our heuristics and new heuristics to correlate email messages to source code.

4. INTEGRATING EVOLUTIONARY ANNOTATIONS INTO ECLIPSE

The proposed architecture consists of a database that contains all the evolutionary annotations that are correlated to the source code. As development artifacts are changed (e.g., source code changes are committed, email messages are sent, defects are reported, etc), the database is updated. A web service links the database and Eclipse. Eclipse requests annotations based on where the developer is working (e.g., a method, a class, a set of files, a project) and updates the user's perspective. Eclipse provides a useful framework for presenting, through views and gutter annotations, the EAs to developers. The plug-in architecture of Eclipse allowed us to create a prototype that integrates EAs into a environment that already contains useful development tools and supports many programming languages and hardware and software platforms.

The screenshot from the EA prototype is shown in Figure 2. The screenshot is based on EAs related to the Apache HTTPd 1.3 source. By selecting the section of code that needs to be understood, the related EAs are shown in the Eclipse EA view. In this case a *diff* was also performed since evolution is often easier to understand when one can see the changes between versions. The oldest EA pertains to a bug reported by a non-core developer. The next EA is an email that contains the review comments and votes of two independent reviewers of the proposed patch. The next EA is a subversion commit log explaining what has been changed. The most recent EA is an email indicating that new problems have been discovered in the code.

In Figure 2, the EAs have been filtered for a particular section of code. The filters could be relaxed to include an entire file or project. Gutter annotations (not shown in the screenshot) are used in a file to indicate specific section of code that contain annotations. Global EAs are indicated as markup on file and project icons. It is also possible to filter EAs based on any of the attributes. For example, it is possible to restrict EAs to a particular version, type (e.g., bugs), and author. This second type of filtering is important because most sections of the code will have at least a commit log associated with them making the guttered annotations cluttered. We are also considering using more advanced filtering techniques, such as allowing users to bookmark or tag multiple sections of code and then base our EA sets on these selections. EAs could complement degree of interest tools like Mylar [4]. Mylar hides files that are not related to the current development task. EAs can use the same model to show users how the code evolved to the current state, helping to inform future changes.

5. FURTHER WORK

This paper describes a research project in progress. One fundamental question about EAs that needs to be addressed is how useful

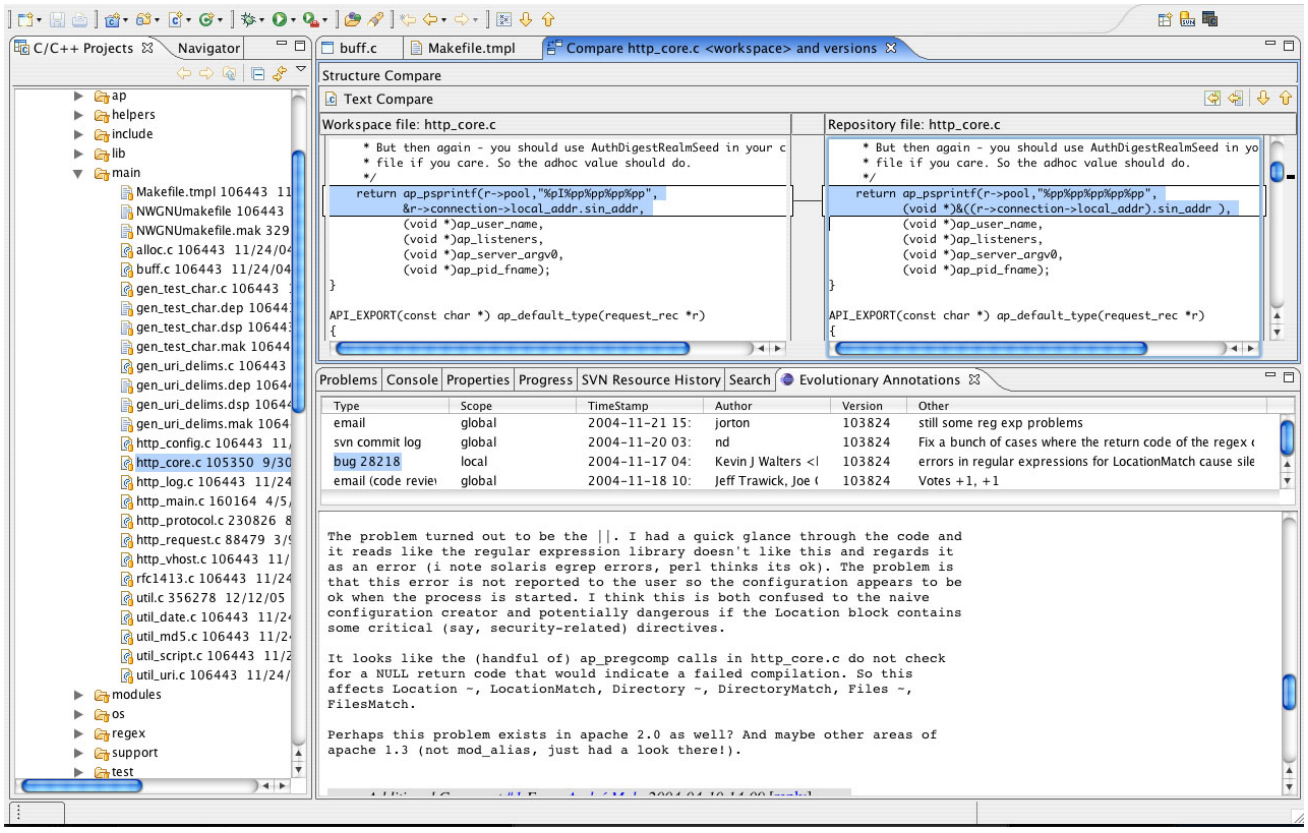


Figure 2: Evolutionary annotations related to a section of highlighted source code.

would they be to assist during program comprehension and maintenance?

Their usefulness will depend, for a given project, on how accurate they are. In other words, we first need methods to accurately evaluate the quality and quantity of evolutionary annotations and how well they can be cross-referenced to the source code they refer to. We also need empirical studies that extract, study and evaluate EAs for a variety of projects.

As with any other type of documentation, some EAs will be of high quality, while others will provide very little insight. Some projects will have a vast number, while others will have very few. Furthermore, from the point of view of a given developer trying to understand the evolution of a given part of the code, what really matters is the quality and number of the annotations to that particular part of the system. Experiments intended to evaluate the usefulness of EAs need to take these factors into account.

Tool support is also needed. It is necessary to create methods and heuristics for the extraction and correlation of evolutionary annotations, in particular for email discussions. It is also necessary to create methods to rank, filter and present evolutionary annotations so they do not overwhelm the developer.

The ideas underlying evolutionary annotations pose many interesting questions for future work and for discussion.

6. REFERENCES

[1] D. Cubranić, 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.

[2] D. M. German. Using software trails to reconstruct the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):367–384, 2004.

[3] D. M. German. An empirical study of fine-grained software modifications. *Journal of Empirical Software Engineering*, 2005. Accepted for publication Sept 25, 2005, to appear in the Special Issue of Best Papers of ICSM 2004.

[4] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[5] P. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS -305-IR, University of Victoria, 2006.

[6] E. Tryggeseth. Report from an Experiment: Impact of Documentation on Maintenance. *Empirical Software Engineering*, 2(2):201–207, 1997.

[7] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

A Study of the Contributors of PostgreSQL

Daniel M. German
Software Engineering Group, Dept. of Computer Science
University of Victoria
dmg@uvic.ca

ABSTRACT

This report describes some characteristics of the development team of PostgreSQL that were uncovered by analyzing the history of its software artifacts as recorded by the project's CVS repository.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Life Cycle, Programming Teams

General Terms

Management

Keywords

Software evolution, mining software repositories.

1. QUESTIONS ADDRESSED

Our goal was to answer the following questions:

1. Who are the contributors to PostgreSQL and what can we know about the number of their contributions?
2. Has the team's composition changed over the years?
3. Can we identify any patches that are submitted by persons without CVS accounts?
4. Do they keep strong territoriality over the code base? In other words, are most files modified by only one developer?
5. Do contributors have different roles? For instance, can identify people who program, create tests cases, document, etc?

2. INPUT DATA AND APPROACH

We used as the main source for our analysis the CVS repository of the project. We proceeded to mine it twice. The first time was Sept 9, 2004. During this stage we proceeded to materialize every revision of every source code file (i.e. we recreated every version of every source code file ever submitted to the repository). The second time was Feb 21, 2005; this time we only retrieved the metadata of the changes to the system. In both cases the first recorded change was made on July 9, 1996. One important point to highlight is that development of PostgreSQL started long before they started using CVS, and therefore, we only have a fraction of the total history of the project. For instance, Release 1.0 was published in 1995, and some copyright notices in some files date back to 1983.

For the mining of the repository we used the framework provided by softChange [2] (softChange uses PostgreSQL as its storage backend). We proceeded to create some derived information:

- We reconstructed atomic commits (in the rest of this paper we will refer to them as Modification Records –MRs)
- We reconstructed every version of every source code file submitted to CVS from July 9, 1996 to Sept. 9,2004.
- We created various statistics for each version of a file, and every MR, such as LOCSs, number of functions added and removed in each revision/MR, whether the revision/MR included only changes to the source code, etc.
- We have found that larger MRs in PostgreSQL tend to be changes in comments or code reorganizations, and if they are considered in any analysis they can add a significant amount of noise (for instance, in PostgreSQL the largest commits are reindentation of the source code –a task performed on a regular basis–or the update of the copyright's year) [1]. For that reason we have selected a subset of MRs (which we call codeMRs). codeMRs satisfy the following conditions: a) they are committed to the main branch of development; b) they contain at least one source file; and c) they contain at most 25 files. We believe that codeMRs are more representative of programming effort compared to MRs, and, in general, using codeMRs instead of MRs improves the quality of any analysis.

3. ANSWERING THE QUESTIONS

3.1 Who are the contributors?

We identified 28 different contributors to PostgreSQL who have a CVS account. Only 4 of them have contributed more than 5 percent of MRs. The proportion of MRs per contributor is depicted in figure 1. Like many other open source projects, most of the commits are done by a handful of individuals.

3.2 Has the team composition changed over the years?

Figure 2 shows, for any given year, the proportion of contributions of MRs for the top 10 contributors. Some observations can be made: the majority of contributions are performed, in any given year, by two persons (which we will call the core team); and one of the early members of the core team (*vadim*) was replaced by (*tgl*) between 1997 and 1998. Nonetheless, the team's composition has been very stable over the years.

3.3 Can we identify any patches that are submitted by persons without CVS accounts?

One problem faced with the analysis of the evolution of a software system based on CVS metadata is the difficulty of identifying contributions by those without a CVS account (these contributions are commonly known as patches). We reviewed the logs of each of

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

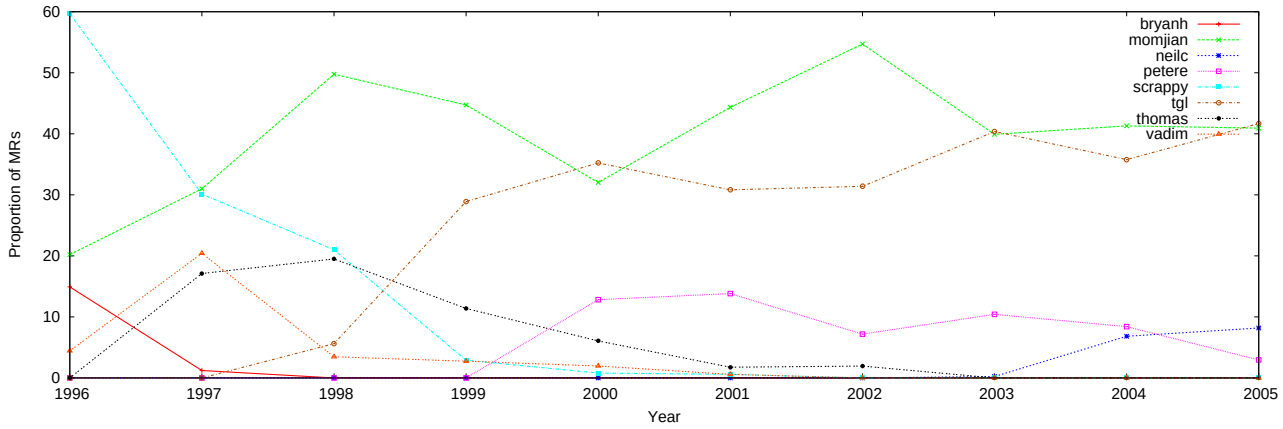


Figure 2: Proportion of contributors of MRs by year

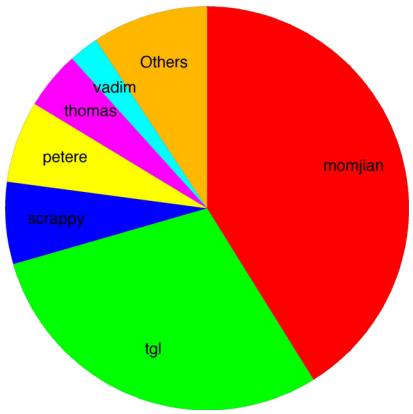


Figure 1: Proportion of contributors of MRs

the 364 codeMRs *momjian* performed during 2005 trying to find any indication of these patches. We were able to identify 110 MRs (roughly 1/3 of the total) to be patches submitted by 46 different individuals. We also found ample evidence of bug reporting by a large number of individuals. It is important to note that the format in which these contributors are acknowledged is different from other projects (at least in the experience of the author): the words "patch" and "contributed" are rarely mentioned, and the email of the developer is not included either. We also inspected some commits by *tgl* to be patches, but they were significantly fewer; but *tgl* committed a large number of MRs where he acknowledges people who submitted bug reports, designs and other contributions.

3.4 Do they keep strong territoriality over the code base?

A change to a file does not necessarily mean somebody has expertise on that file. This observation is best exemplified when the source code of PostgreSQL is reindented (a process that is done on a regular basis, usually before a release) or, at the beginning of a new year, when the copyright statement at the top of each file is changed. The person who reindents the file might not have any idea of what the code being reindented does. For that reason we decided to study changes to files in codeMRs. Furthermore, territoriality might change over time, thus we concentrated in changes performed during 2005. We proceeded to compute, for each pair (contributor, directory):

$$T_c^d = \frac{revisions\ by\ contributor\ c\ in\ directory\ d}{total\ revisions\ to\ directory\ d}$$

During 2005, 173 directories were modified (by a total of 10 people). We found that 123 of these directories had one developer responsible for at least 70% of the changes ($T_c^d \geq 0.7$). In 81 of these directories (primarily in the database engine) the responsible was *tgl* ($T_{tgl}^d \geq 0.7$) . The next was *momjian* with 18 directories; it should be taken into account that *momjian* is responsible for committing patches submitted by contributors without a CVS account (as previously discussed) and therefore he might not have created those modifications (but he probably reviewed them, nonetheless).

3.5 Do contributors have different roles?

We have already discussed that *momjian* is responsible for applying patches, and *tgl* is responsible for most of the source code. Other observations are: *petere* has been responsible for committing most of the internationalization files (.po), while some CVS account holders have taken care of translating PostgreSQL into languages they know (for example *alvherre*, who has committed Spanish translations, or *dennis*, Swedish).

4. CONCLUSIONS

At first we were surprised by how small and stable over the years the core team of PostgreSQL has been. Its CVS repository shows that, in the last years, only two persons have been responsible for most of the source code. We needed to inspect the history of the project in more detail, and were surprised to learn that there is a very large number of contributors who send source code *patches* to the project. This is an important lesson for anybody trying to inspect the history of projects, particularly when the analysis is done automatically. In the end we learned that PostgreSQL has a large and vibrant community who contributes bug reports and patches.

5. REFERENCES

[1] D. M. German. An empirical study of fine-grained software modifications. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 316–325, Sept 2004.

[2] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softChange. In *Proceedings SEKE 2004 The 16th Internation Conference on Software Engineering and Knowledge Engineering*, pages 336–341, June 2004.

Co-Change Visualization Applied to PostgreSQL and ArgoUML*

(MSR Challenge Report)

Dirk Beyer
EPFL, Switzerland

ABSTRACT

Co-change visualization is a method to recover the subsystem structure of a software system from the version history, based on common changes and visual clustering. This paper presents the results of applying the tool CCVisu, which implements co-change visualization, to the two open-source software systems PostgreSQL and ArgoUML. The input of the method is the co-change graph, which can be easily extracted by CCVisu from a Cvs version repository. The output is a graph layout that places software artifacts that were often commonly changed at close positions, and artifacts that were rarely co-changed at distant positions. This property of the layout is due to the clustering property of the underlying energy model, which evaluates the quality of a produced layout. The layout can be displayed on the screen, or saved to a file in SVG or VRML format.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement — *Restructuring, reverse engineering, and reengineering*

General Terms: Design

Keywords: Software visualization, software clustering, software structure analysis, force-directed graph layout

1. METHOD

In reverse engineering and reengineering, we often want to extract a description of the system structure from available resources. Even if a structure description (often 'as-designed') is available, it can be useful to complement it by an extracted description of the 'as-build' structure. *Co-change visualization* is a method that extracts such a description, and aims to help in reverse engineering and re-engineering activities like understanding the structure of the system, change impact and change propagation analysis, coupling analysis, architecture and design quality analysis.

Approach and tool used. The approach of co-change visualization is introduced in [2], and implemented in the tool CCVisu [1]. It requires as input the version history, which is almost always available, and automatically produces a visualization that groups together components that were often changed together, and separates independent components.

Input data. For the two example systems, we take as input the version log information, as (in case of Cvs) obtained by applying the command `cvs log -Nb` (only default branch, ignore tags). We consider the whole development period from project start to Feb. 8, 2006 (extraction date). From this input, we extract the *co-change graph on file level*. The nodes in the (bipartite, undirected) co-change graph are files and commits. An edge between a file node f and a commit node c exists if file f was changed by commit c. The table below presents the characteristics of the graphs. For the details of the method and **related work** we refer to [2, 1].

System	Files	Commits	Changes	Log file
PostgreSQL	4,125	20,500	88,468	17 MB
ArgoUML	10,142	10,137	57,091	16 MB

2. RESULTS AND EVALUATION

The commit nodes and the edges are omitted in the visualizations for readability. The file nodes were drawn in different colors, to compare the grouping suggested by the layout with an authoritative decomposition, according to documentation. The area of a circle is proportional to the number of changes of the file. Each layout was computed within 5 min on a 1.7 GHz Pentium M laptop, using only 200 iterations of the minimizer. The layouts in SVG or VRML format provide (interactively) the file names as annotation and basic zoom features. The figures in this paper are annotated with the names of the subsystems (gray boxes). The layouts in SVG and VRML format, the Cvs log files, and the co-change graphs in RSF, are available on the supplementary web page at http://mtc.epfl.ch/∼beyer/ccvisu_msr.

PostgreSQL. In the authoritative decomposition we considered 12 subsystems of PostgreSQL, and assigned a color to each subsystem: executor (red), optimizer (blue), parser (cyan), storage (magenta), catalog/commands/nodes (yellow), access (dark cyan), port (olive-green), regression test (brown), interfaces (light blue), include (light green), utilities (light gray), and documentation (green).

We can use the colors to evaluate if CCVisu has positioned the 4,125 files in groups in agreement with the authoritative decomposition. Figure 1 clearly separates the main clusters of the documentation (top right, largest circle at bottom is TODO file), the interfaces for libpg (center right) and ecpg (bottom right), and the regression test files (top left) from the backend files (center left), and from the include and utilities (center). To get more insights into the backend files on the left (here not separated), we need to 'zoom' into this part by restricting the co-change graph to the backend, and computing a new layout for this subgraph.

Figure 2 visualizes the backend only. The subsystems that formed the large group on the left in Fig. 1 are now better

*This research was supported in part by the MICS NCCR of the SNSF.

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

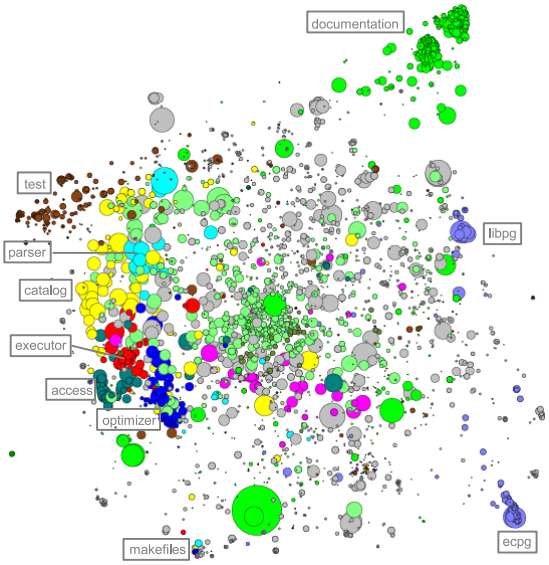


Figure 1: PostgreSQL

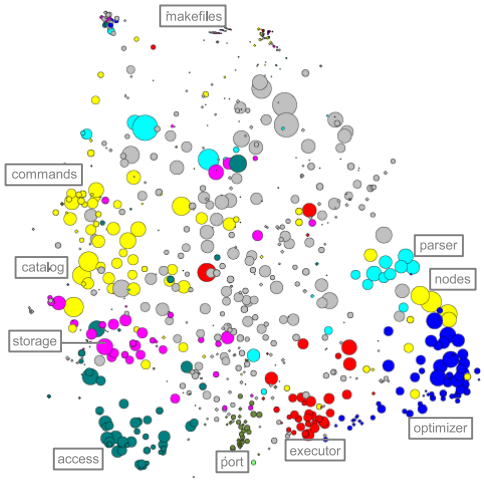


Figure 2: PostgreSQL — backend only

separated. The layout separates from the rest the executor, port, and access subsystems (bottom). It puts the optimizer, nodes, and parser into one group (right), but does not merge the three groups, which makes sense according to the authoritative decomposition. The commands, catalog, and storage files (left) are not separated but also not merged. The gray nodes blur the otherwise clear picture: they represent the utility files, which are used by all subsystems, and therefore they are placed correctly by the method. The three groups containing files in every color at the top are three collections of makefiles — since they necessarily change together, they are placed together although they are assigned to different subsystems in the authoritative decomposition.

ArgoUML. The authoritative decomposition divides the files into 9 parts: old development files (uci in green, uci-gef in brown), documentation (yellow), test files (blue), cognitive (cyan), diagrams (magenta), UI (dark cyan), UML-UI (red), and model (light blue). The files that could not be assigned to any subsystem are drawn in light gray (consisting of utilities, makefiles, configuration files, etc.).

The placement of the 10,142 files of ArgoUML is shown in Fig. 3. The old development branch is clearly separated (top right). Also, the www documentation files are shown

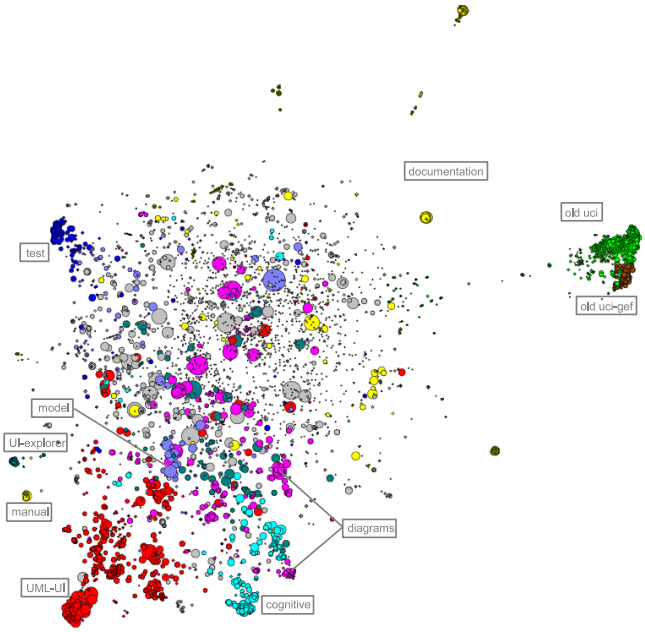


Figure 3: ArgoUML

as several clusters (right side), and some implementation-dependent parts of the documentation (e.g., cookbook, config) are placed close to the corresponding source files. Furthermore, the test files (top left) are nicely separated from the rest. The files for the UML-UI (almost completely) form the red clusters at the bottom, and also the files of the 'cognitive' subsystem are separated. The cluster with the most files of the UI subsystem is the explorer, which is separated from the rest on the very left (close to the manual cluster).

The diagrams and model files are spread over the picture. A restriction of the visualization to the source files, as done for PostgreSQL, leads to a picture (not shown here) where the diagrams subsystem is separated, but the model subsystem does still not form a cluster because this subsystem is responsible for interfacing and exchanging data. For example, the largest circle (light blue) is the class ModelFacade. The visualization allows the following interpretation: the subsystems for UML-UI, cognitive, diagrams, and the explorer component of the UI subsystem are reasonably loosely coupled, and the rest is dependent on many other components (expected to be necessary for UI, models, parts of diagrams). We omit a more detailed discussion for space.

Conclusion. The resulting visualization provides the software engineer with valuable information, e.g., for reverse engineering it reveals the subsystem structure, for program understanding it illustrates which artifacts depend on each other, and for quality assessment it can be used to highlight unstable parts of the system. That the co-change graph is indeed a good prediction model can be shown by comparing two layouts that result from splitting the co-change data into a (virtual) past and future. The method relates not only source code files, but also, e.g., SQL query files, test files, and documentation files, to program source code files.

3. REFERENCES

[1] D. Beyer. Co-change visualization. In *Proc. ICSM'05, Industrial and Tool volume*, pages 89–92, 2005.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proc. IWPC*, pages 259–268. IEEE, 2005.

Mining Software Repositories with CVSgrab

Lucian Voinea
Technische Universiteit Eindhoven
The Netherlands
l.voinea@tue.nl

Alexandru Telea
Technische Universiteit Eindhoven
The Netherlands
alext@win.tue.nl

ABSTRACT
In this paper we address the process and team analysis categories of the MSR Mining Challenge 2006. We use our CVSgrab tool to acquire the data and interactively visualize the evolution of ArgoUML and PostgreSQL, in order to answer three relevant questions. We conclude summarizing the strong and weak points of using CVSgrab for mining large software repositories.

Categories and Subject Descriptors
D.2.7 [**Software engineering**]: Distribution, Maintenance, and Enhancement – *documentation*, *reengineering*;

General Terms
Management, Measurement, Documentation

Keywords
Evolution visualization, Software visualization, CVS

1. INTRODUCTION
The MSR Mining Challenge brings together researchers and practitioners in the field of software repository mining, and stimulates them to compare their tools and approaches. To establish a common ground for comparison, two benchmarking datasets are proposed: the ArgoUML and PostgreSQL CVS repositories. ArgoUML is an open source project with a history of 6 development years, 4452 evolving files, contributed by 37 authors. PostgreSQL is an open source project with a history of 10 development years, 2829 evolving files, contributed by 27 authors. We used our CVSgrab tool [1] from the Visual Code Navigator toolset [2] to analyze the process and the team structure of these projects. The process and findings are described below.

2. SETUP
CVSgrab [1] is a tool for visualizing the evolution of large software projects. CVSgrab includes mechanisms to query CVS repositories locally or over the Internet. File contents are retrieved on demand and cached locally, which massively speeds up the mining process. CVSgrab can detect and cluster files with similar evolution patterns, using several evolution similarity metrics [1]. Unlike classical CVS clients such as WinCvs or TortoiseCVS, CVSgrab provides extensive support for interactively showing evolutions of huge projects on a single screen, with minimal browsing. Figure 1 depicts the architectural pipeline of CVSgrab:

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

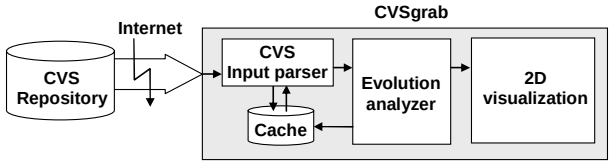


Figure 1: CVSgrab architectural pipeline

CVSgrab uses a simple 2D layout (see Figures 2,3,4): Each file is drawn as a horizontal strip, made of several segments. The *x*-axis encodes time, so each segment corresponds to a given version of its file. Color encodes version attributes, e.g. author, type, size, release, presence of a given word in the version's CVS comment, etc. Atop of color, texture may be used to indicate the presence of a specific attribute for a version. File strips can be sorted along the *y*-axis in several ways, thereby addressing various user questions.

3. RESEARCH QUESTIONS
We used CVSgrab to acquire, analyze and visualize the evolution information for ArgoUML and PostgreSQL. We formulated a number of relevant team and process related questions and tried to answer them using CVSgrab's interactive visual mechanisms:

Q1: What is / was the development process?

Assessing the development process is important for project and/or process auditors. Usually, the assessment outcome is based on developer interviews and not on the real situation. We propose using CVSgrab to base such assessments on the real data in CVS. We used CVSgrab to visually compare the development process behind both ArgoUML and PostgreSQL (Figure 2). We sorted the files in the increasing order of their creation time. We used color to encode file type: In Figure 2 left, documentation files are yellow (HTML) and light green (images) and Java sources are red. In Figure 2 right, C sources are blue, C headers are light green, test suites are red, and documentation files are green.

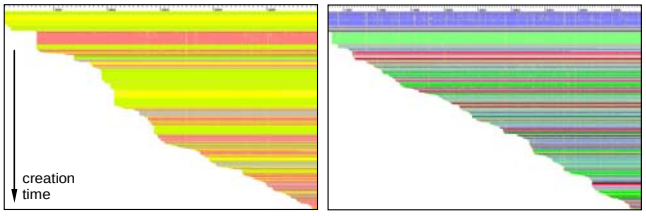


Figure 2: Evolution of file type: ArgoUML (left), PostgreSQL (right). Creation time increases from top to bottom.

We now easily see that the development of ArgoUML started with some documentation files (yellow, light green), possibly containing the system specification and/or design. Implementation source files followed only later. For a significant period, i.e. more than 1/3 of the development time, no new source files appear. This suggests the system architecture was stable in this period.

Next, source and documentation files are alternatively added in large chunks, suggesting a coarse iterative development process with few architectural changes. In contrast, the development of PostgreSQL starts directly with a set of C source files, followed shortly after by a set of header files (light green). This suggests the system, as present in CVS, was not developed from scratch, but started atop of some previous project. However, the system specification / design either does not exist, or it is not maintained: There are just a few documentation files (green) and these appear much later in the project. The system architecture appears to be less stable, as header files containing interfaces and corresponding implementation files are added throughout the entire project. The set of committed files is frequently interrupted by test suites (red). This suggests an iterative development process in which added functionality is tested before implementing new one.

Q2: What are the main contributors and their responsibilities?

During the development and maintenance of large software projects, new developers often join and/or leave the team. It is very important that newcomers quickly get familiar with the rest of the development team and their responsibilities. In Figure 3, we used the same file layout as in Figure 2 to show the evolution of the two projects. However, color encodes now the ID of the developers, so Figure 3 shows the evolution of contributions.

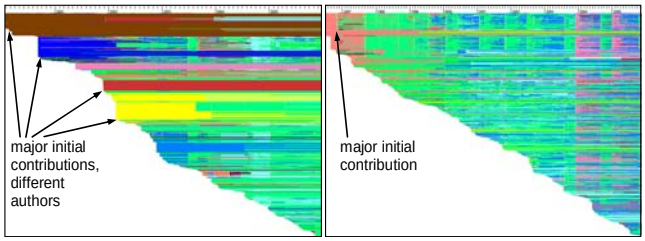


Figure 3: Evolution of author contributions: ArgoUML (left), PostgreSQL (right)

We can see that, both for ArgoUML and PostgreSQL, there is only one author for each major initial contributions, i.e. areas with a steep slope of the time curve. However, these contributions might represent the work of more developers, initially committed by one configuration manager. The evolution of PostgreSQL reveals another interesting pattern: alternative contribution of two developers, e.g. green and blue vertical stripes for the middle period of evolution. The responsibilities of the two developers are however different. We can see that the contributions of the 'green' author involve many files simultaneously, while the 'blue' author commits fewer files, but more often. This suggests the 'green' author has rather the role of a configuration manager that applies formatting changes to the entire code (e.g. indentation), while the 'blue' author affects the system functionality in small increments.

Q3: Where are located the development issues discovered and solved during alpha testing of some given release?

To track errors during debugging, it is of paramount importance to narrow down the location of the code introducing the fault. This might not always coincide with the location where the program crashes. Moreover, an error might be caused by the resolution of another issue. In such situations, it is useful to easily identify the code that changes from one system release to another *and* in the same time addresses a given issue. In Figure 4, we used the same file layout as in Figure 2 to show the evolution of ArgoUML. Color encodes versions that belong to a given system release: light green = VERSION_0_14_ALPHA_1, dark cyan = VERSION_0_14_F, red = both releases, grey = none. Grainy texture shows versions that contain a reference to the word "issue" in their associated CVS comment file. We see that only a few files that have been changed during the alpha testing of release VERSION_0_14 appear to reference the word "issue". This is shown as light green horizontal segments followed by textured dark cyan ones. At close inspection, we saw that all this code refers to ArgoUML's parsing mechanism.

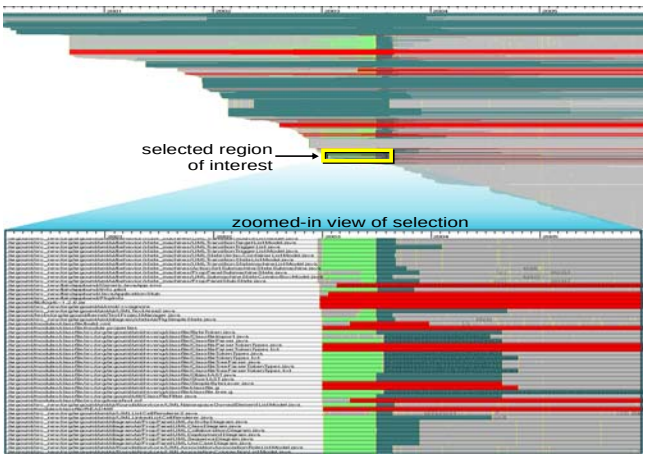


Figure 4: Identifying ArgoUML version changes between release VERSION_0_14_ALPHA_1 and release VERSION_0_14_F that contain the word "issue" in their commit comment. Inset shows a zoomed-in region, for better insight.

4. DISCUSSION

We have briefly illustrated the use of the CVSgrab tool [1] for process and team analysis of large software projects. We used as input data the MSR Challenge 2006 projects: ArgoUML and PostgreSQL. The presented use cases confirmed us that CVSgrab has a very good scalability: It can give comprehensive evolution overviews for projects of thousands of files and hundreds of versions, thus meeting industry size requirements. CVSgrab can easily answer questions that involve the formation of a large uniform color pattern, e.g. Q1 and Q2 in this paper, or questions involving comparison of a small number of colors, e.g. Q3. Secondly, the tight integration of the on-demand, Internet-based CVS data browsing, acquisition, and visualization in CVSgrab massively simplified the process of getting quick overviews of huge projects. Finally, CVSgrab can be easily extended to support different scenarios, by adding different file sorting techniques, attribute-to-color mappings, and file similarity metrics, yielding a powerful CVS mining tool. A complete version of the CVSgrab tool is available for download on the Visual Code Navigator home page at: http://www.win.tue.nl/~lvoinea/VCN.html

References

[1] Voinea, L., Telea, A. CVSgrab: Mining the History of Large Software Projects. *Proc. EUROVIS 2006*, ACM Press, 2006, to appear.

[2] Lommerse G., Nossin F., Voinea S.L., Telea A.: The Visual Code Navigator: An Interactive Toolset for Source Code Investigation. *Proc. IEEE InfoVis*, IEEE Press, 2005, 24 – 31

Mining Additions of Method Calls in ArgoUML

Thomas Zimmermann[1] Silvia Breu[2] Christian Lindig[1] Benjamin Livshits[3]

[1] Dept. of Computer Science
Saarland University
Saarbrücken, Germany
{tz, cl}@st.cs.uni-sb.de

[2] University of Cambridge
Computer Laboratory
Cambridge, UK
silvia@ieee.org

[3] Dept. of Computer Science
Stanford University
Stanford, USA
livshits@cs.stanford.edu

ABSTRACT

In this paper we refine the classical co-change to the addition of method calls. We use this concept to find usage patterns and to identify cross-cutting concerns for ArgoUML.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*; D.2.9 [**Management**]: Software configuration management

General Terms

Management, Measurement

1. INTRODUCTION

One of the most frequently used techniques for mining version archives is *co-change*. We specialize this concept to the *addition of method calls*:

> **Two method calls that are added together in the same transaction, are related to each other.**

We use the concept of *co-additions* for the following two tasks:

- *Find usage patterns*, such as "the methods `containsNode` and `containsEdge` are frequently called together."
- *Identify cross-cutting concerns*, such as "the first statement of every method calls the `info` method to log the method name."

In Section 2 we will describe our input data and the tools we used; we present our results for usage patterns in Section 3 and for cross-cutting concerns in Section 4.

2. INPUT DATA AND TOOLS

We applied our mining techniques to the ArgoUML repository that was supplied for the MSR challenge [4]. We restricted our analysis to the `src_new` directory that contains the actual source code of ArgoUML. All data was collected with an extended version of the eROSE plug-in [2] for the ECLIPSE environment. For mining, we used SQL queries and the Xelopes data mining library [5].

To reconstruct transactions we use the *sliding window* approach with a window size of 200 seconds. For each transaction we compute the set of *newly added method calls*. For this we compare the total set of method calls from the actual and the previous transaction. The total set of method calls is computed for each transaction by traversing the abstract syntax trees of all affected files.

For a call expression $c_1().c_2(). \ldots c_n()$ we only take the final method call $c_n()$ into account. Since we only analyze one file at a time, the full signature for method c_n isn't available. Instead, we augment it with the number of parameters, as shown in Table 1. Analyzing single files rather than complete snapshots makes our preprocessing cheap, as well as platform- and compiler-independent.

Pattern	Count
localize(2) addField(2)	57
localize(1) lookupIcon(1)	45
addCaption(4) addField(4)	43
addButton(1) lookupIcon(1)	41
localize(1) addField(2)	28
findFigsForMember(1) findType(1)	23
addModelEventListener(2) removeModelEventListener(2)	**19**
addModelEventListener(3) removeModelEventListener(3)	**13**
addFocusListener(1) addKeyListener(1)	**12**
hasMoreElements(0) nextElement(0)	**12**
error(2) debug(1)	11
addSeperator(0) addField(2)	10
info(1) isInfoEnabled(0)	**10**
max(2) isDisplayed(0)	9
containsNode(1) containsEdge(1)	**8**

Table 1: Usage patterns for ArgoUML.

3. MINING USAGE PATTERNS

Our approach is based on an observation: Method calls that are added to source code simultaneously often represent a pattern. To identify such patterns, we performed *frequent pattern mining* on the set of added method calls.

We focused our analysis on *intra-procedural* patterns: patterns that occur within a single method. In terms of mining this means that we do not use complete transactions as input but group transactions by the method in which a call was added. Furthermore, we ignored calls to frequently used JAVA methods, such as `iterator`, `hasNext`, and `toString`, since patterns involving these methods are well-known.

Table 1 shows the patterns we mined, sorted by decreasing frequency. Actual usage patterns are printed in boldface, thus the precision is 40%. Below we discuss a few examples.

- `addModelEventListener`, `removeModelEventListener`
  This pattern is used when elements are changed. First, the listener is removed for the old element, then the element is changed, and finally the listener is added for the new element.

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

```
if (Model.getFacade().isAElement(target)) {
   Model.getPump().removeModelEventListener
      (this, target);
}
target = t;
if (Model.getFacade().isAElement(target)) {
   Model.getPump().addModelEventListener
      (this, target, "name");
}
```

- `addFocusListener`, `addKeyListener`
  This pattern indicates a relationship between the focus and a key listener: A user may enter text only to graphical elements that are in focus.
- `isInfoEnabled`, `info`
  Sometimes the return value of `isInfoEnabled` is checked before the `info` method is called.

```
if (LOG.isInfoEnabled()) {
   LOG.info("Removing feature " + feature);
}
```

- `containsNode`, `containsEdge`
  These two methods are frequently called with the same arguments to check whether an edge is valid; if not, an error is logged.

```
if (!containsNode(destModelElement)
      && !containsEdge(destModelElement)) {
   LOG.error("some message");
   return false;
}
```

4. MINING CROSS-CUTTING CONCERNS

Programs can be modularized in only one way at a time. Aspect-oriented programming (AOP) remedies this by factoring out aspects and weaving them back in a separate processing step. For existing projects to benefit from AOP, these cross-cutting concerns must be identified first. This task is called *aspect mining*.

Our hypothesis is that *not all cross-cutting concerns exist from the beginning, but some emerge over time*. By analyzing where developers add code to a program, we can identify cross-cutting concerns. Our approach searches transactions for sets of locations L where at each location calls to a set of methods M have been added. In other words: The calls to methods M are spread throughout source code locations L. We call such a pair (M, L) an *aspect candidate*. In order to identify aspect candidates that actually cross-cut a considerable part of a program, we ignore all candidates (M, L) where $|L| < 7$ or $|M| \cdot |L| < 20$. This means that each aspect candidate has to cross-cut at least 7 locations, and it has to comprise at least 3 method calls that got added.

For ArgoUML we identified 230 aspect candidates in 73 out of 6,286 transactions. Below we discuss a few examples.

Logging. We observed that the transaction with the log message *"Replaced deprecated log4j Category with Logger."* inserted several calls to methods `debug`, `error`, and `warn`. The last two methods turned out to be false positives. However, for `debug` we found several cross-cutting calls that logged the method names as shown in the following example:

```
public void doAction(int oldStep) {
   LOG.debug("doAction " + oldStep);
   ...
}
```

This logging could have easily been realized with an aspect.

Illegal arguments. The transaction with the log message *"Made the methods look a little more alike. Collected the numerous IllegalArgument calls in methods. [...]"* inserted many cross-cutting calls to `illegalArgument` or one of its variants. These calls are always last in the method body:

```
public String getValueOfTag(Object handle) {
   if (handle instanceof MTaggedValue) {
   return ((MTaggedValue) handle).getValue();
   }
   return illegalArgumentString(handle);
}
```

In this case the method `illegalArgumentString` throws an `IllegalArgumentException` and returns a `null` object. Most of the 262 calls to `illegalArgument` methods could have been realized as aspects.

Instance of a thing. The transaction with the log message *"Replace every single instance of something instanceof MThing with ModelFacade.isAThing(something)"* inserted many `isA` calls to the source code. `isA` methods look as follows:

```
public boolean isAClassifier(Object handle) {
   return handle instanceof MClassifier;
}
```

There exist 111 methods of the above form; these methods could have easily been generated with aspects.

In our previous work [1] we showed that mining cross-cutting concerns from version archives has a high precision, for the top 20 aspect candidates of ECLIPSE we reached up to 90%. Measuring recall requires knowing all aspect candidates, which is typically only possible for a few small benchmark projects.

5. CONCLUSION

Co-addition of method calls identifies usage patterns; a usage pattern may be actually a cross-cutting concern when all locations where calls were added call the same set of locations. Both usage pattern and cross-cutting concerns can be identified by mining version archives, as demonstrated by the ones we found in ArgoUML.

Usage patterns and cross-cutting concerns have several benefits. Mining usage patterns can locate defects in software and supports program understanding. Knowing cross-cuttings concerns helps to reduce maintenance effort and is the prerequisite for refactoring a legacy system into an aspect-oriented design.

For a more detailed description of our mining approaches, we refer to our publications on finding usage patterns [3] and identifying cross-cutting concerns [1].

6. REFERENCES

[1] S. Breu and T. Zimmermann. Mining Aspects from History. Submitted for publication.

[2] eROSE. Guiding Programmers in Eclipse. http://www.st.cs.uni-sb.de/softevo/erose/.

[3] V. B. Livshits and T. Zimmermann. Dynamine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proc. Europ. Software Engineering Conf./ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 2005.

[4] MSR. Mining Challenge 2006. http://msr.uwaterloo.ca/challenge/.

[5] Prudsys AG. XELOPES Library. http://www.prudsys.com/Produkte/Algorithmen/Xelopes/.

Using Software Birthmarks to Identify Similar Classes and Major Functionalities

Takeshi Kakimoto Akito Monden Yasutaka Kamei
Haruaki Tamada Masateru Tsunoda Ken-ichi Matsumoto
Nara Institute of Science and Technology
8916-5 Takayama Ikoma Nara Japan 630-0192
{takesi-k, akito-m, yasuta-k, harua-t, masate-t, matumoto}@is.naist.jp

ABSTRACT
Software birthmarks are unique and native characteristics of every software component. Two components having similar birthmarks indicate that they are similar in functionality, structure and implementation. Questions addressed in this paper include: Which are similar class files? Can they be gathered into one class file? What are major functionalities among class files? To answer to these questions, this paper analyzed the similarity of birthmarks for all pairs of classes in ArgoUML, and visualized them using Multi-Dimensional Scaling (MDS). As a result, three pairs of very similar class files, which seem to be made by copy-and-paste programming, were identified. Also, four major functionalities were identified in the MDS space.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *Product Metrics*; K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance;*

General Terms: Measurement, Experimentation

Keywords
software birthmark, multi-dimensional scaling

TARGET OSS PROJECT
ArgoUML (written in Java)

MINING AREA
- Change impact, propagation coupling analysis
- Architecture and design quality analysis

MINING QUESTIONS
The following two questions are addressed in this paper.

(1) Which are similar class files?

This question needs to be answered when one wants to refactor a Java program. Similar class files are often refactored into one class file to improve software maintainability. Also, when one modifies a class file, he/she often needs to find similar class files that need to be modified as well.

(2) What are major functionalities among class files?

This question needs to be answered when one joins a project and tries to understand the mapping between class files and their functionalities.

INPUT DATA
From 1,432 class files of ArgoUML release 0.20, 61 classes having more than 30 lines of source code were chosen as an input dataset.

1. APPROACH AND TOOLS USED

1.1 Birthmark
Java birthmarks[1] are unique and native characteristics of every Java class files. Originally, birthmarks are used to detect the stolen (i.e. illegally copied) Java class files across two different projects. In this paper we use birthmarks to find similar class files in a project to help maintenance activities.

We used a Java birthmark tool called *jbirth*[1] to extract four types of birthmarks from each class file: (1) constant values in field variables (*CVFV birthmark*), (2) sequence of method calls (*SMC birthmark*), (3) an inheritance structure (*IS birthmark*), and (4) used classes (*UC birthmark*). Two class files having similar birthmarks indicate that they are similar in functionality, structure and implementation.

To compute the similarity between two class files p and q in terms of their birthmarks, we used the following definition [1].

Definition (Similarity) Let $f(p) = (p_1, \ldots, p_n)$ and $f(q) = (q_1, \ldots, q_n)$ be birthmarks with length n, extracted from class files p and q. Let s be the number of pairs (p_i, q_i)'s such that $p_i = q_i$ $(1 \le i \le n)$. Then, similarity between $f(p)$ and $f(q)$ is defined by: $s/n \cdot 100$.

1.2 Multi Dimensional Scaling (MDS)
After computing the similarity of birthmarks for all pairs of 61 class files, we used MDS to visualize their relationships. Major functionalities can be identified as clusters in the MDS space. We used SPSS as a MDS tool.

Table 1. Pair of classes having high similarity birthmarks.

Class pairs	similarity
uml.ui.behavior.collaborations.PropPanelCollaboration	96.09
uml.ui.behavior.use_cases.PropPanelUseCase	
uml.ui.behavior.collaborations.PropPanelMessage	93.91
uml.ui.behavior.state_machines.PropPanelTransition	
uml.ui.foundation.core.PropPanelClass	92.78
uml.ui.foundation.core.PropPanelAssociationClass	

[1] http://se.naist.jp/jbirth/

MSR'06, May 22-23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

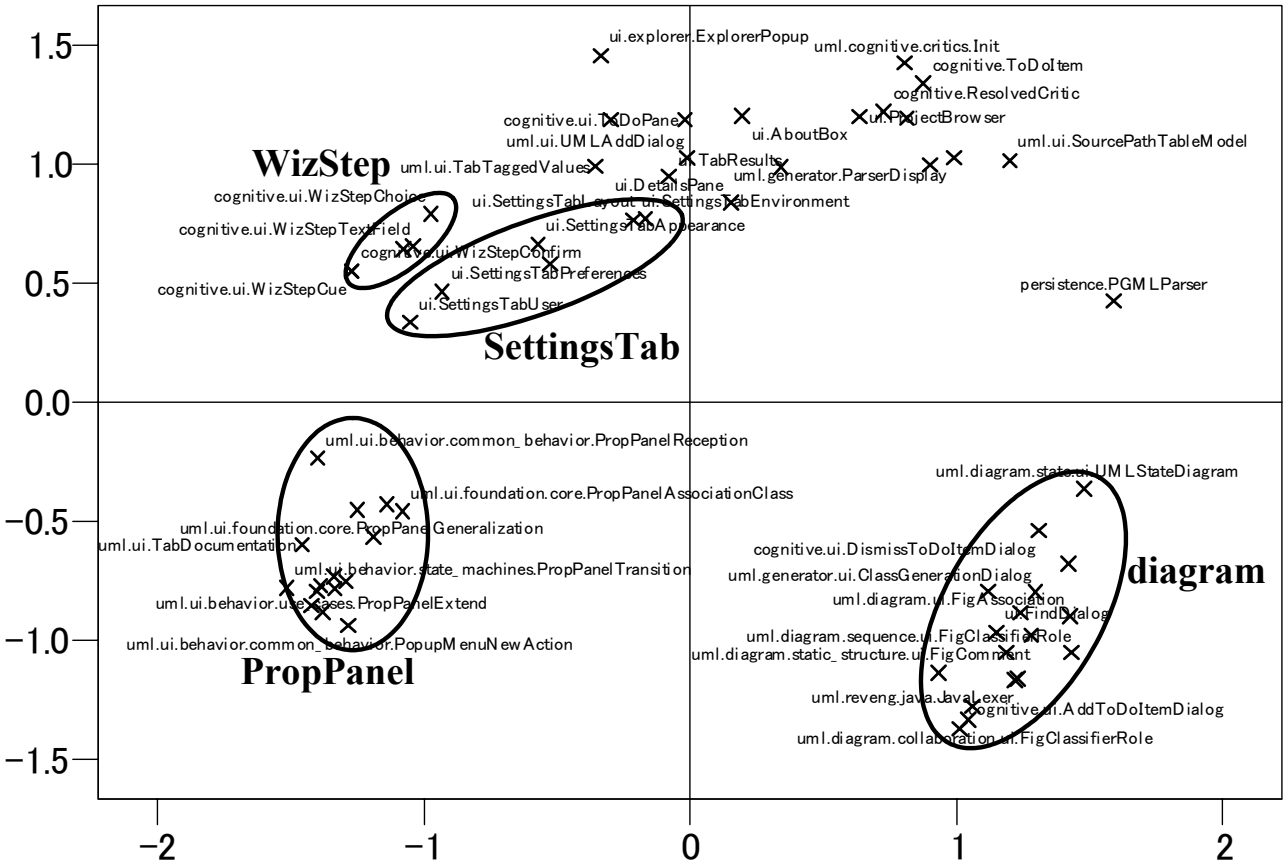


Figure 1. Relationship among class files in MDS space based on similarity of birthmark.

2. RESULTS AND INTERPRETATIONS

2.1 Finding Similar Class Files

Table 1 shows pairs of classes that had high-similarity birthmarks (similarity > 0.9). As we investigated their source code, each pair had very similar functionality, structure and implementation. It can be considered that these pairs were made by copy-and-paste programming, and can be refactored so as to reduce the duplicated code.

2.2 Finding Major Functionalities

Figure 1 shows relationship among classes in the MDS space. Classes having similar birthmarks are located in near space, and classes having dissimilar birthmarks are located far apart. From Figure 1, we could identify the following four major functionalities.

- Most of classes at the lower right part ("diagram" circle) were related to diagram (e.g. FigAssociation class, FigUseCase class.)
- Classes at the lower left part ("PropPanel" circle) were related to PropPanel (e.g. PropPanelAttribute class, PropPanelOperation class.)
- Classes related to WizStep were in "WizStep" circle (e.g. WizStepConfirm class.)
- Classes related to SettingsTab were in "SettingsTab" circle (e.g. SettingsTabEnvironment class.)

All these functionalities were identified by finding clusters and similar file names in the MDS space. We believe that using birthmarks together with MDS is useful to understand the relations between class files and to roughly recognize their functionalities.

3. CONCLUSIONS

This paper analyzed the similarity of birthmarks for all pairs of classes in ArgoUML, and visualized them using MDS. As a result, three pairs of very similar class files were identified. Also, four major functionalities (diagram, PropPanel, WizStep and SettingsTab) were identified in the MDS space.

4. ACKNOWLEDGMENTS

This work is supported by the EASE (Empirical Approach to Software Engineering) project of the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

5. REFERENCES

[1] Tamada, H., Nakamura, M., Monden, A., Matsumoto, K. Java birthmark –Detecting the software theft. *IEICE Transactions on Information and Systems, E88-D*, 9 (Sept. 2005), 2148-2158

How Long Did It Take To Fix Bugs?

Sunghun Kim, E. James Whitehead, Jr.

University of California,
Santa Cruz, CA, USA
{hunkim, ejw}@cs.ucsc.edu

ABSTRACT

The number of bugs (or fixes) is a common factor used to measure the quality of software and assist bug related analysis. For example, if software files have many bugs, they may be unstable. In comparison, the bug-fix time—the time to fix a bug after the bug was introduced—is neglected. We believe that the bug-fix time is an important factor for bug related analysis, such as measuring software quality. For example, if bugs in a file take a relatively long time to be fixed, the file may have some structural problems that make it difficult to make changes. In this report, we compute the bug-fix time of files in ArgoUML and PostgreSQL by identifying when bugs are introduced and when the bugs are fixed. This report includes bug-fix time statistics such as average bug-fix time, and distributions of bug-fix time. We also list the top 20 bug-fix time files of two projects.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance.*

General Terms

Management, Measurement

1. INTRODUCTION

The number of bugs is commonly used to measure software quality. For example, if a file has 100 cumulative bugs over its development history, we may assume the file is more instable than one that had no bugs in its history. We believe that both bug counts and bug-fix times are important factors for bug related analysis. We can determine the bug-fix time by identifying bug-introducing changes (fix-inducing changes [5]) and corresponding bug fixes. The bug-fix time can be used to measure software quality. For example, if bugs in a software file take a long time to be fixed, it may indicate the file is instable or we need to pay more attention to the file.

We compute the bug-fix time of two open source projects, ArgoUML (period 1/2002 - 3/2003) and PostgreSQL (period 07/1996-11/2000), and report bug-fix time statistics. Our goal is to demonstrate how bug-fix time can be used as a factor for bug related analysis.

MSR '06, May 22-23, 2006, Shanghai, China.

2. EXPERIMENT SETUP

To compute bug-fix time, we need to identify bug-introducing changes and their corresponding fixes, and then measure the time between them. For example, suppose a bug was introduced (in file '*foo*') at revision 3 and it was fixed at revision 9 as shown in Figure 1. We compute the bug-fix time by subtracting the commit time of revision 3 from that of revision 9.

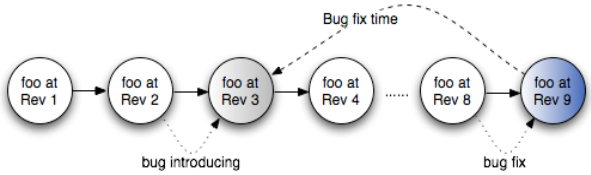


Figure 1. Bug-fix time example.

We first extract change histories of the two projects using the Kenyon infrastructure [1]. We next identify bug fixes by mining change logs. There are two ways to identify a bug-fix: searching for keywords such as "Fixed" or "Bug" [4] and searching for references to bug reports like "#42233" [2, 3, 5]. We use the keyword-based change log search to identify bug fixes. We identify bug-introducing changes by applying the fix-inducing change identification algorithms described in [5]. We then obtain the commit time of the identified bug-introducing changes and their corresponding bug fixes from project histories. From the commit times, we compute each bug-fix time and the average bug-fix time of each file.

3. BUG-FIX TIME

In this section we report bug-fix time statistics of two projects.

3.1 Bug Numbers and Fix Time

We show the distribution of bug counts for each bug-fix time in Figure 2 and Figure 3. Bug fixes times in buggy files range from 100-200 days (the spikes in Figure 2 and Figure 3).

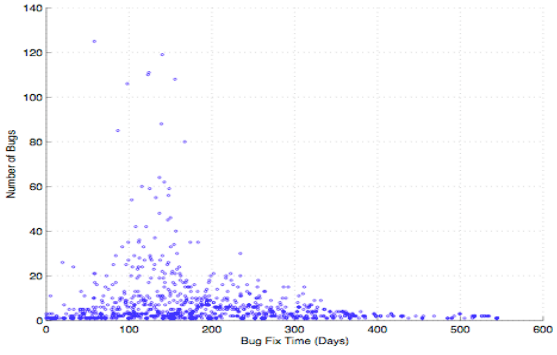


Figure 2. Distributions of bug counts by bug-fix time of ArgoUML.

Table 1. Top 20 files with greatest bug-fix times

Rank	ArgoUML Files	Bug fix time (days)	Bug count	PostgreSQL Files	Bu fix time (days)	Bug count
1	argouml/src_new/org/argouml/uml/ui/UMLInitialValueComboBox.java	332	9	pgsql/src/backend/commands/define.c	504	19
2	argouml/src_new/org/argouml/uml/ui/UMLAttributesListModel.java	328	6	pgsql/src/backend/access/rtree/rtree.c	482	14
3	argouml/src_new/org/argouml/ui/NavigatorConfigDialog.java	324	9	pgsql/src/backend/utils/hash/dynahash.c	474	17
4	argouml/src_new/org/argouml/kernel/ProjectMember.java	320	7	pgsql/src/backend/utils/cache/inval.c	472	16
5	argouml/src_new/org/argouml/uml/ui/UMLTaggedBooleanProperty.java	318	7	pgsql/src/include/storage/bufpage.h	450	14
6	argouml/src_new/org/argouml/uml/ui/ActionSaveGraphics.java	317	6	pgsql/src/backend/utils/cache/relcache.c	444	84
7	argouml/src_new/org/argouml/uml/ui/UMLMultiplicityComboBox.java	317	6	pgsql/src/backend/catalog/pg_proc.c	425	18
8	argouml/src_new/org/argouml/uml/cognitive/critics/WizAssocComposite.java	315	6	pgsql/src/backend/optimizer/path/allpaths.c	422	37
9	argouml/src_new/org/argouml/ui/FindDialog.java	312	7	pgsql/src/backend/executor/nodeMergejoin.c	419	17
10	argouml/src_new/org/argouml/uml/DocumentationManager.java	312	15	pgsql/src/backend/utils/fmgr/dfmgr.c	408	17
11	argouml/src_new/org/argouml/uml/ui/ActionNew.java	310	12	pgsql/src/backend/commands/trigger.c	408	25
12	argouml/src_new/org/argouml/cognitive/ui/ToDoPerspective.java	306	6	pgsql/src/backend/utils/cache/catcache.c	407	32
13	argouml/modules/php/src/org/argouml/language/php/generator/GeneratorPHP.java	305	11	pgsql/src/backend/utils/init/postinit.c	399	46
14	argouml/src_new/org/argouml/uml/cognitive/critics/CrNameConflict.java	305	6	pgsql/src/backend/executor/nodeHash.c	393	19
15	argouml/src_new/org/argouml/uml/ui/UMLComboBoxEntry.java	304	6	pgsql/src/backend/executor/nodeAgg.c	391	53
16	argouml/src_new/org/argouml/cognitive/critics/ui/CriticBrowserDialog.java	304	8	pgsql/src/backend/rewrite/rewriteDefine.c	385	29
17	argouml/src_new/org/argouml/uml/ui/ActionAddOperation.java	292	15	pgsql/src/backend/access/gist/gist.c	384	19
18	argouml/src_new/org/argouml/uml/ui/ActionDeleteFromDiagram.java	289	10	pgsql/src/backend/nodes/readfuncs.c	382	60
19	argouml/src_new/org/argouml/uml/ui/ActionAddTopLevelPackage.java	289	6	pgsql/src/backend/catalog/pg_type.c	376	22
20	argouml/src_new/org/argouml/language/ui/SettingsTabNotation.java	287	15	pgsql/src/backend/commands/rename.c	376	24

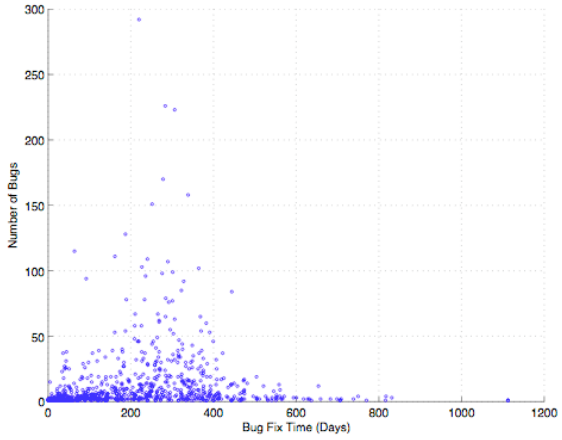


Figure 3. Distributions of bug counts per bug-fix time of PostgreSQL.

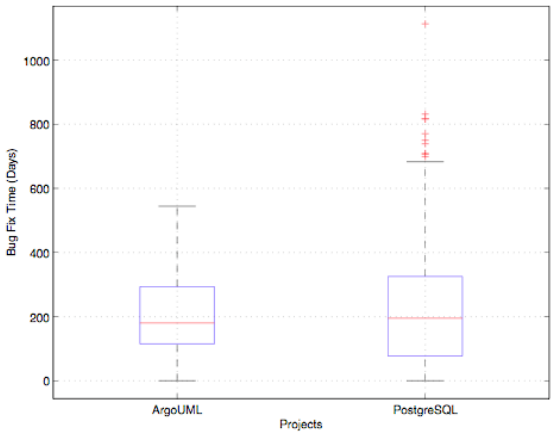


Figure 4. Bug-fix time (days) of the two projects. Two boxes indicate 50% of bug-fix time (25% to 75% quartile). The middle line in boxes indicates the median value of bug-fix time.

Figure 4 shows the bug-fix time of the two projects using box plots. They show that fixing 50% of the bugs requires appx. 100 to 300 days (the two boxes in Figure 4). The median bug-fix time is about 200 days.

3.2 Number and Bug-fix Time

Table 1 lists the top 20 files with greatest bug-fix times, whose bug counts are greater than average. The listed files may need attention to determine why bug fixes take such a long time and may need to be refactored to permit faster bug fixes in the future .

4. CONCLUSION

By mining software histories of two projects, ArgoUML and PostgreSQL, we computed and analyzed the bug-fix time of each file. We believe that bug-fix time is useful, and should be widely used for bug related analysis.

5. REFERENCES

[1] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[2] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," Proc. of 25th International Conference on Software Engineering (ICSE), Portland, Oregon, pp. 408-418, 2003.

[3] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of 2003 Int'l Conference on Software Maintenance (ICSM'03), pp. 23-32, 2003.

[4] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," Proc. of International Conference on Software Maintenance (ICSM 2000), San Jose, California, USA, pp. 120-130, 2000.

[5] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 24-28, 2005.

Mining Refactorings in ARGOUML

Peter Weißgerber, Stephan Diehl
University of Trier
Computer Science Department
54286 Trier, Germany
weissger@uni-trier.de, diehl@acm.org

Carsten Görg*
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
goerg@cc.gatech.edu

ABSTRACT

In this paper we combine the results of our refactoring reconstruction technique with bug, mail and release information to perform process and bug analyses of the ARGOUML CVS archive.

Categories and Subject Descriptors: D.2.8[Software Engineering]:Metrics; D.2.5[Software Engineering]:Testing and Debugging

General Terms: Algorithms, Management, Measurement.

Keywords: Refactoring, mails, bugs, evolution, re-engineering.

1. INTRODUCTION

In this study we mine the CVS archive of ARGOUML for refactorings that have been performed during the development and evolution of ARGOUML. We relate the refactorings of each day to the number of overall changes on that day to detect both phases with many and phases with almost no refactorings. We look especially at the phases before major release dates, because this may help the project manager in planning pre-release phases, or to plan release dates ahead.

To see if refactorings in ARGOUML have an effect on the occurrence of new bugs and on communication between the developers, we relate the refactorings to bug reports in ISSUEZILLA respectively to mails on the developer mailing list. If the error rate would increase with the refactoring ratio, the project manager would have to enforce the use of automated refactoring tools, or the used refactoring tools or methods may be poor.

Finally, we examine if there are incomplete refactorings which could possibly lead to errors. Mining changes for such incomplete refactorings can uncover bugs that have been introduced long ago.

2. MINING REFACTORINGS IN ARGOUML

2.1 Computing the Refactoring Ratio

In [1] we introduced our technique to reconstruct refactorings from software archives such as CVS. For each day, we determine which blocks (fields, methods in a class) have been changed and which of these are affected by refactorings. Thus we get values for the following metrics:

Normalized number of changed blocks (per day):
$\%CB_d = \frac{\#CB_d}{\#CB_{max}}$
where $\#CB_{max} = max\{\#CB_d | d \text{ day in project's lifetime}\}$

Number of refactorings per changed block (per day):
$\%RB_d = \frac{\#RC_d}{\#CB_d}$ where $\#RC_d$ is the number of non-overlapping, disambiguated refactoring candidates for day d

2.2 Computing Bug and Mail Ratios

To determine if days with a high refactoring ratio result in fewer errors than other days, we look at the number of bugs filed per day in the ISSUEZILLA system of ARGOUML. As developers usually do not detect errors immediately after the program change that caused them, we compute the number OB_d of all new *defects filed within the next five days* (which roughly approximates a working week). For each day we relate this value to the number of changes:

Normalized number of bugs per changed block:
$\%BB_d = \frac{\#OB_d}{\#CB_d \#OB_{max}}$
where $\#OB_{max} = max\{\#OB_d | d \text{ day in project's lifetime}\}$

Additionally, we are interested in whether refactorings have an effect on the amount of communication between the developers. Therefore, we consider the development mailing list of ARGOUML and count for each day the number AM_d of archived mails. We relate this number to the number of changes as follows:

Normalized number of mails per changed block:
$\%MB_d = \frac{\#AM_d}{\#CB_d \#AM_{max}}$
where $\#AM_{max} = max\{\#AM_d | d \text{ day in project's lifetime}\}$

3. ARGOUML RESULTS

3.1 Process Analysis: The Pre-Release Phase

Figure 1 shows the time periods before and after the four release dates of the stable series of ARGOUML from 2002 until 2005. In all cases, we see the same pattern:

- Before the release dates, there seem to be testing phases where only few changes have been done at all, but many new bug reports have been opened.
- In each case after the testing phase and immediately before the release, changes with high refactoring ratio have been performed.

*The author was supported by a fellowship within the Postdoc-Program of the German Academic Exchange Service (DAAD).

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

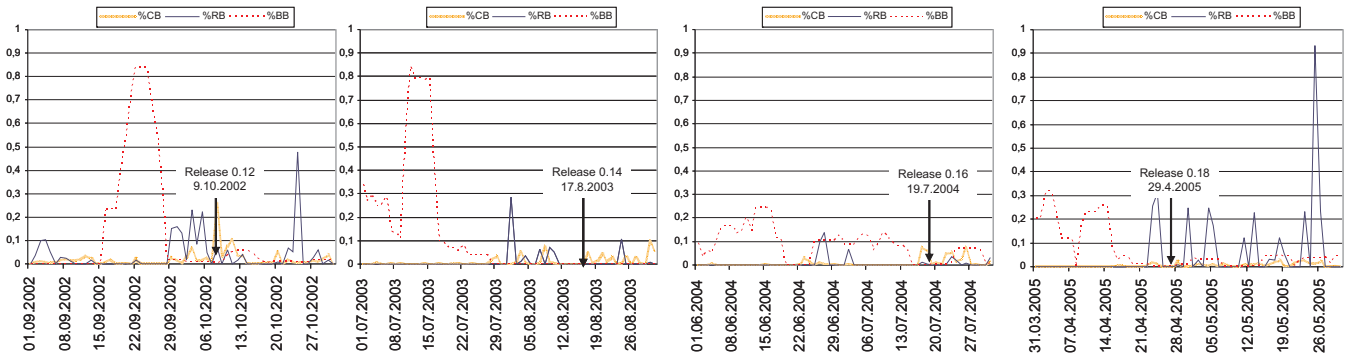


Figure 1: Relative number of changes, refactorings and bugs before major releases.

3.2 Bug Analysis: Correlation between Refactorings, Mails, and Bugs

Figure 2 shows the values of the normalized number of bugs per day $\%BB_d$, as well as the normalized number of mails per day $\%MB_d$ compared to the refactoring ratio per day. While the Spearman correlation between $\%RB$ and $\%BB$ is only about 0.3, it stands out that after days with a high refactoring ratio only few bug reports have been opened in the bug tracking system. The same holds for mails: When the refactoring ratio is high, few mails have been written. However, we are aware that these correlations could be accidental or caused by other factors like feature freezes that we did not yet take into account.

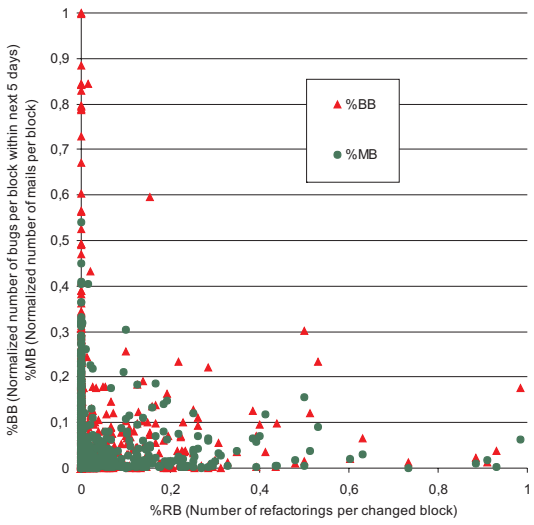


Figure 2: Few bug reports and mails after days with high refactoring ratio.

3.3 Bug Analysis: Incomplete Refactorings

Refactoring reconstruction can also be used to detect incomplete, and thus erroneous refactorings [2]. In these cases, parameters have been added to or removed from methods, but the developer did not change the corresponding methods in super-, sub-, or sibling classes accordingly. In ARGOUML we found 33 transactions containing such incomplete refactoring candidates between Jan 2003 and Dec 2005.

Figure 3 shows a candidate for a possibly incomplete refactoring: the sibling classes `ActionSaveGraphics` and `ActionSaveProjectAs` both contained the method `trySave(bool)`. The refactoring *RemoveParamter* was applied only to the method in the class `ActionSaveGraphics` and possibly it also should be applied to the method in the class `ActionSaveProjectAs`.

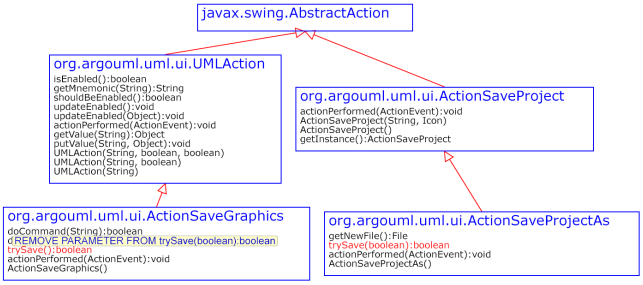


Figure 3: Missing Remove Parameter refactoring.

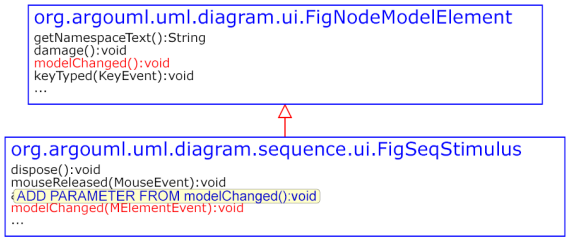


Figure 4: Missing Add Parameter refactoring.

Figure 4 shows the application of an *AddParameter* refactoring to the method `modelChanged()` in the class `FigSeqStimulus`. The refactoring has not been applied to the method `modelChanged()` in its superclass `FigNodeModelElement`. However, some transactions later the *AddParameter* refactoring has been applied to the method `modelChanged()` in the superclass and also to methods in five other subclasses of `FigNodeModelElement`. Apparently the refactoring was incomplete in the beginning.

Acknowledgments. Michael Stockman kindly provided the bug data for ARGOUML.

4. REFERENCES

[1] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of International Workshop on Program Comprehension (IWPC05)*, St. Louis, Missouri, USA, May 2005.

[2] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *Proceedings of International Workshop on Mining Software Repositories (MSR05)*, St. Louis, Missouri, USA, May 2005.

Applying the Evolution Radar to PostgreSQL

Marco D'Ambros, Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland
{marco.dambros, michele.lanza}@lu.unisi.ch

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Maintenance, Version Control, Re-engineering, Reverse Engineering

General Terms: Measurements, Design.

Keywords: Evolution, Logical Coupling, Visualization.

1. GOALS

In this report we describe the results of the application of our approach, the Evolution Radar [2], on the PostgreSQL system. The mining questions we want to answer are:

1. What are the relationships among the system modules in terms of logical coupling? How are these relationships characterized? Which are the main responsibles for the logical couplings, *i.e.*, the best candidates for starting a reengineering process?

2. How have these relationships evolved over time? When have refactorings been applied on the modules? In which phase is the system in the current version?

2. INPUT DATA

To analyze the target system, *i.e.*, PostgreSQL, we use its whole history, as recorded by the CVS version control system, stored in a database called Release History Database (RHDB) [1, 3]. The database populating process, performed in batch mode, consists in (i) doing a checkout of the system, parsing it and storing the structure information in the database, (ii) parsing the CVS logs and storing all the commit-related information. The RHDB includes information about all the files in the system,*i.e.*, source code, documentation, make-files, *etc.* For our analysis we consider only the source code data, *i.e.*, `.c` and `.h` files (since PostgreSQL is written in c). We decompose the system using the top-most directories in the `src` directory tree, *i.e.*, we define a module as all the files belonging to a directory subtree.

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

3. THE EVOLUTION RADAR APPROACH

The Evolution Radar (see Figure 1) visualizes the logical coupling of one module with the others (see [2] for details). The module in focus is placed in the middle of a pie chart, where each sector represents one of the other modules. The size of each sector depicts the size of each module in terms of number of files. The modules are sorted according to this size metric.

The files of each of those modules are represented as circles and placed (and colored) according to the logical coupling they have with the module placed in the center. The closer the files are to the center (the hotter, from blue to red, the color is), the more coupled they are.

Given a module M, a file f, and a time interval (t_1, t_2), we define the logical coupling between the two as:

$$LC(M, f, t_1, t_2) = \max_{f_i \in M} \{sc(f_i, f, t_1, t_2)\} \quad (1)$$

where $sc(f_i, f, t_1, t_2)$ is the number of shared commits (performed at the same time with a tolerance of 200 seconds) between f_i and f during the time interval (t_1, t_2)[1].

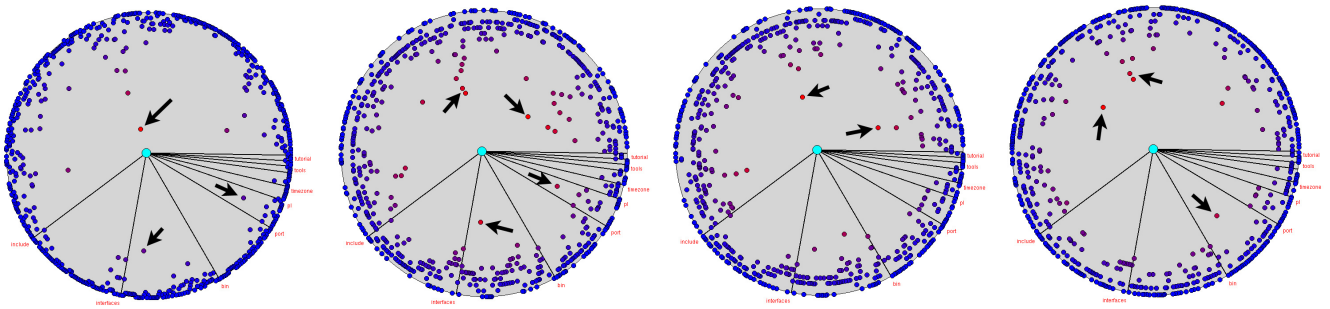
4. RESULTS

We consider the three biggest modules with respect to the number of files: backend (673 files), include (394 files) and interfaces (84 files). For each module we build four Evolution Radars (using the module as the center of the radar) corresponding to the last four years of development of the module. Then we study the relationships of the target module with the five other biggest modules in the system with respect to the logical coupling information. For this study we both analyze the view and compute some measures characterizing the evolution of the couplings. In details we define:

- **Strength (*s*)**: The total value of the logical coupling between the target module M_t and another module M (a slice). It is equal to the sum of the logical coupling of all the files of M with M_t.

- **Distribution (*d*)**. The percentage of files involved in the logical coupling. It is equal to the number of files of M having a logical coupling with M_t divided by the total number of files of M.

[1]We don't need a normalized value, *i.e.*, weighting the logical coupling with the number of commits, because we study the evolution of the logical couplings, thus we compare absolute values of logical couplings over time instead of analyzing one value only.



(a) 2002 (b) 2003 (c) 2004 (d) 2005

Figure 1: Evolution Radars for the backend module from 2002 to 2005.

- **Outliers (*ol*)**. The files of M having a logical coupling with M_t much higher than all the others. Those are detected directly on the view instead of using a formal definition.

Figure 1 shows the four Evolution Radars for the backend module (where the arrows highlight the outliers), while the computed measures and the detected outliers are listed in Table 1.

		2002	2003	2004	2005
include	*s*	832	928	957	431
	d	53%	73%	71%	45%
	ol	parsenodes.h	parsenodes.h nodes.h bufmgr.h	parse-nodes.h guc.h	parsenodes.h nodes.h executor.h
interfaces	*s*	81	161	107	95
	d	31%	81%	63%	55%
	ol	tabcomplete.c	tabcomplete.c	none	none
bin	*s*	105	212	183	91
	d	58%	84%	78%	64%
	ol	none	none	none	none
port	*s*	31	63	70	37
	d	30 %	62%	53%	38%
	ol	none	none	none	path.c
pl	*s*	36	52	38	32
	d	40%	45%	50%	40%
	ol	pl_exec.c	pl_exec.c	none	none

Table 1: Results for the backend module.

Conclusions on the Backend Module. As we can see from Figure 1 and Table 1 the backend module was initially (2002) logically decoupled from all the other modules but the include one. For this module the distribution value was relatively low (53%) and the coupling was mainly due to outliers, mostly `parsenodes.h`. In the following year the logical coupling with all the other modules increased, for both strength and distribution, implying that the quality of the design of the backend module decreased as well.

In 2004 we observe that: (i) the dependency with include stayed stable at high values of strength and distribution, (ii) the logical coupling with interfaces and pl decreased and (iii)`tabcomplete.c` and `pl_exec.c` which were outliers up to this moment were not outliers any more. We deduce that a refactoring phase was previously (2003) applied for these two modules (interfaces and pl). This is one of the reasons of the high logical coupling values in the previous year.

In the last year the dependencies with all the other modules decreased (for both strength and distribution), especially with include, bin and port. We deduce that in 2004 the backend module was refactored, since the logical coupling decreased for all the other modules.

Suggestions for the Backend Module. The files `parsenodes.h` and `nodes.h` of the include module should be further analyzed and, in case, moved to the backend module. The first was an outlier from 2002 to 2005 while the second in 2002 and 2005. They were coupled with backend for a long time and they were still coupled in the last year. The file `path.c` of the port module and `executor.h` of the include module should also be analyzed. They were only recently coupled with backend, implying that the dependencies are due to recent changes. We suggest to analyze them because an early refactoring is less expensive.

5. CONCLUSION

The Evolution Radar allows us to study the evolution of the dependencies among system modules and to detect the outliers, the best starting points for the refactoring process. It is also helpful to understand if the dependencies of the outliers are due to recent changes or they were coupled with the target modules for many years. We have shown the results of the application of our approach on the biggest module of PostgreSQL. We have found candidates for reengineering and refactoring phases in the evolution of the modules. For the other two biggest modules we have found similar results but we have not presented them for lack of space.

6. REFERENCES

[1] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationships. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages xxx–xxx. IEEE CS Press, Mar. 2006.

[2] M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Proceedings of MSR 2006 (International Workshop on Mining Software Repositories)*, pages xxx–xxx, May 2006.

[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Los Alamitos CA, Sept. 2003. IEEE Computer Society Press.

Examining the Evolution of Code Comments in PostgreSQL

Zhen Ming Jiang and Ahmed E. Hassan
Software Architecture Group (SWAG)
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
{zmjiang, aeehassa}@uwaterloo.ca

ABSTRACT
It is common, especially in large software systems, for developers to change code without updating its associated comments due to their unfamiliarity with the code or due to time constraints. This is a potential problem since outdated comments may confuse or mislead developers who perform future development. Using data recovered from CVS, we study the evolution of code comments in the PostgreSQL project. Our study reveals that over time the percentage of commented functions remains constant except for early fluctuation due to the commenting style of a particular active developer.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Documentation.*

General Terms
Human Factors.

Keywords
Software Evolution, Software Maintenance, Code comments.

1. INTRODUCTION
Most of the software development effort is devoted to software maintenance. Developers spend about half of their time trying to understand code [1]. Most developers agree that it is not easy to read other people's code. A well documented program is easy to follow and improves the quality of the software [3]. However, in large software systems, due to unfamiliarity with the system or due to time constraints or maybe just laziness, developers are likely to change source code without updating its associated comments. This is a potential time bomb, since outdated comments are misleading and cause confusion. We believe it is worthwhile for managers to monitor the evolution of code comments over time.

We study source code comments in the PostgreSQL project over time. Our focus is on the comments associated with functions. We categorize code comments into two types: *Header Comments* and *Non-Header Comments*. *Header Comments* are comments before the declaration of a function; whereas *Non-Header Comments* are all other comments residing in the body of a function or trailing the function. Developers usually use Header Comments to describe the purpose of a function, and to document its parameters and interfaces. Non-Header Comments are usually used to document algorithms and low level design decisions.

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

Research by Perry *et al*. has shown that at least 66% of bugs in large projects are due to interface errors [4]. Uncommented interfaces or interfaces with outdated comments are likely to cause bugs. In this paper, we examine whether the percentage of functions with header comments (FH) drops over time relative to the functions with non-header comments (FNH). We believe that a drop may indicate that developers are not updating the interface documentations.

2. DISCUSSION ABOUT OUR FINDINGS
To perform our study, we used the C-REX extractor [2] to recover all CVS changes for PostgreSQL from 1996 to 2005. C-REX is able to track the addition and removal of functions and function dependencies over time. It also tracks all changes to comments associated with these functions.

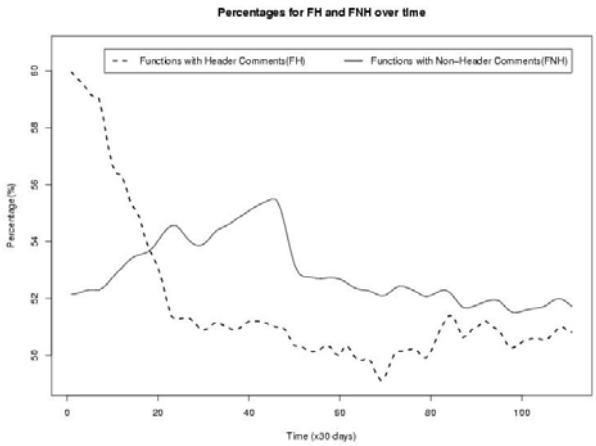


Figure 1: Percentage of FH and FNH Over Time.

Figure 1 shows the percentage of functions with header comments (*FH*) and non-header comments (*FNH*) for every 30 days period. This Figure reveals that:

1. During the initial two year period (the first 30*25 days), there is a steady decrease in the percentage of FH and an increase in the percentage of FNH.

2. After the initial two year period, the percentage of FH and FNH remain steady, and are around 51% and 52%, respectively.
The first finding is worth investigating since it may be due to the removal of many FH or the addition of a large amount of FNH relative to FH. It is also possible that quite a few FH had their header comments removed. The addition of many FNH is troublesome since the interfaces for these functions are not documented and may cause future bugs.

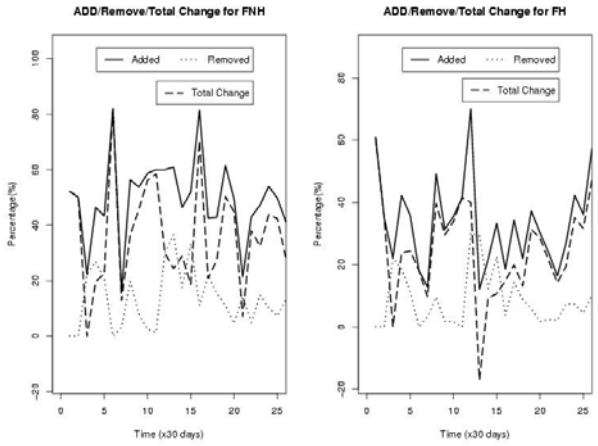


Figure 2: Percentage of Added/Removed/Total Change for FNH and FH.

To investigate the changes in the percentage of FH and FNH, we plot the percentage of addition and removal of FH and FNH during the first two year period in Figure 2. The Figure as well shows the total change (percentage of added – percentage of removed) over time. In both subgraphs in the Figure, we note that the total change line is always above zero (except one case around 13 in the right subgraph). Therefore, we can conclude that more FNH and FH are added than removed during this two year period.

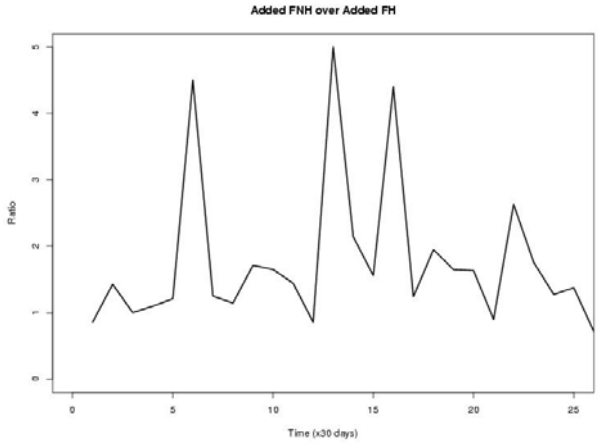


Figure 3: Ratio of Number of Added FNH over Added FH

We now compare the amount of added FNH against the amount of added FH. Figure 3 shows the relationship between added FNH and FH for the two year period. The Figure plots the ratio of added FNH over added FH for every 30 days. We see that the ratio always stays above 1. This indicates that there are always more FNH being added than FH during the initial two year period.

Using the recovered C-REX data which tracks all changes to the source code and the name of the developers who performed these changes, we examine closely the spikes in Figure 3. Our investigation reveals that these spikes are due to a particular developer who contributed a large number transactions during these time periods. These transactions added mainly utility functions to PostgreSQL. The developer has a particular commenting style, where he appends the name of a function at the end of the function's declaration block. For example, in revision 1.13 of the file "./postgres/pgsql/src/backend/utils/adt/geoops.c", he adds a small uncommented utility function called "int4 text".

```
text * int4_text ( int32 arg1 )
{
. . .
} /*  int4_text ( )   */
```

If this method were added by other developers, it would probably become a function with no comments at all; however, in this case it belongs to the category of FNH functions.

3. CONCLUSION AND FUTUREWORK

Correct and up to date comments aid developers in understanding the source code; wrong or outdated comments mislead developers and cause the introduction of bugs. Thus, it is important that managers monitor code comments over time. In this paper, we studied comments in PostgreSQL. We discovered that apart from the initial fluctuation due to the introduction of a new commenting style; the percentage of functions with header and non-header comments remains consistent throughout the development history.

In the future, we plan to investigate the relationship between the decrease in comment rate and the introduction of bugs.

4. REFERENCES

[1] R. Fjeldstad and W. Hamlen. Application program maintenance-report to our respondents. In *Tutorial On Software Maintenance,* pages 13–27.1983.

[2] A. E. Hassan and R. C. Holt. C-REX: An Evolutionary Code Extractor for C. May 2004.

[3] D. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279 – 287, Sorrento, Italy, May 1994.

[4] D. E. Perry and W. M. Evangelist. An Empirical Study of Software Interface Faults—An Update. In *Proceedings of the 20th Annual Hawaii International Conference on Systems Sciences*, pages 113–136, Hawaii, USA, Jan. 1987

Analyzing OSS Developers' Working Time Using Mailing Lists Archives

Masateru Tsunoda
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
masate-t@is.naist.jp

Akito Monden
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
akito-m@is.naist.jp

Takeshi Kakimoto
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
takesi-k@is.naist.jp

Yasutaka Kamei
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
yasuta-k@is.naist.jp

Ken-ichi Matsumoto
Nara Institute of Science and Technology
Kansai Science City, 630-0192 Japan
matumoto@is.naist.jp

Categories and Subject Descriptors
K.6.1 [**Management of Computing and Information Systems**]: Project and People Management – *Staffing;*

General Terms: Management

Keywords
Overtime work, workload

1. INTRODUCTION
We chose PostgreSQL, a relational database system for the MSR mining challenge.

Our research question is in the following mining area:

- Process analysis

Our mining question is "when OSS developers work?" OSS developers' working time may be a good indicator to understand the development style of a project. (For example, if many developers work in office hour, these might be daily works in a company.)

2. INPUT DATA
We used mailing lists (MLs) archives of PostgreSQL, downloaded from http://www.postgresql.org/community/lists/. The MLs mainly consist of user lists and developer lists. We used developer lists archive since we needed developers' working time. Table 1 explains details of each ML. Figure 1 shows amounts of messages of each ML in the developer lists. Amounts of messages were increasing year by year. The ML of hackers had many more messages than other MLs. We extracted MLs archives till December 2005. Note that most of committers' messages were automatically generated when source code was checked into software configuration management repository.

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

Table 1. Description of Each ML

List	Description	Archived from
committers	Notification of CVS commits are sent to this list.	April 2000
hackers	Discussion of current development issues, problems and bugs, and proposed new features.	January 1997
patches	Patches for new features and bug fixes should be sent to this list.	June 2000
www	Discussion of development and coordination of the PostgreSQL websites.	August 2003

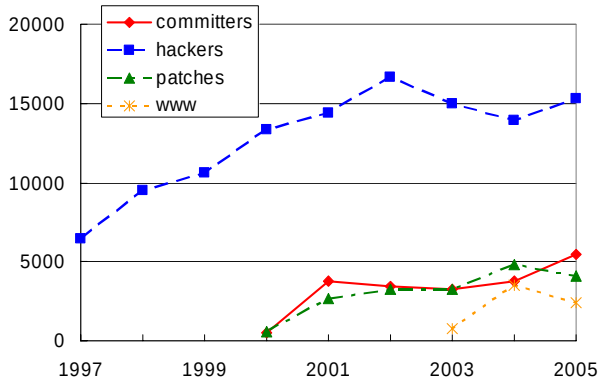


Figure 1. Amount of Messages in Each Year

We picked up "mail sent time" to identify developers' working time. Getting mail sent time from the MLs archives consists of the following two steps: First, we downloaded the MLs archives with

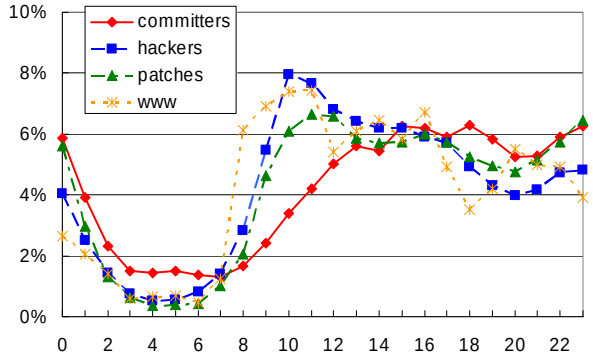


Figure 2. Ratio of Messages in Each Hour

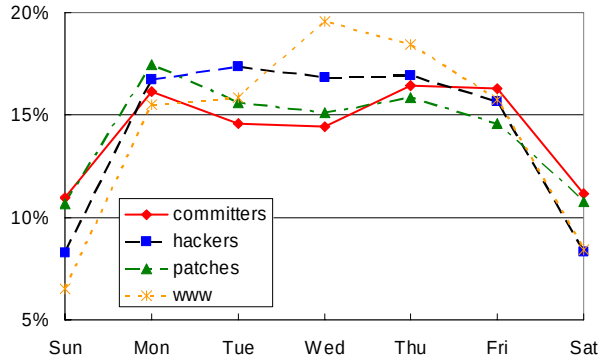


Figure 3. Ratio of Messages in Each Day of a Week

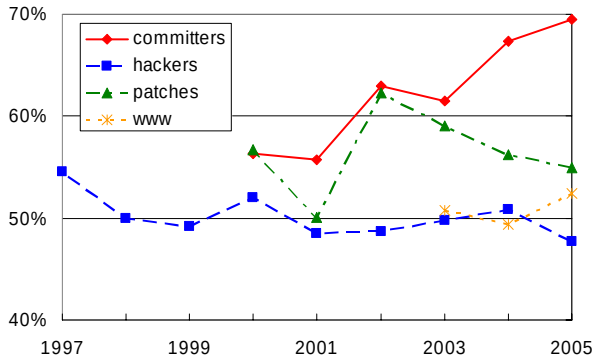


Figure 4. Ratio of Messages Sent at Overtime Period in Each Year

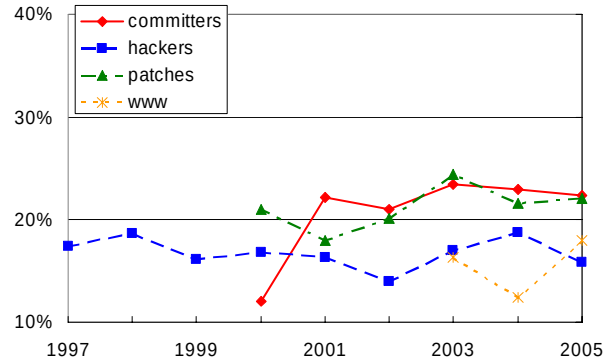


Figure 5. Ratio of Messages Sent on Weekend in Each Year

Irvine[1], a web download tool. Then, we extracted mail sent time from the downloaded archives with a Perl script.

In our analysis, we mainly focused on the following aspects:

- Mail sent hour
- Days of a week of mail sent date
- Difference in hours and date among ML groups
- Time trend

To see developer workload, we defined the overtime period. Overtime period includes before 9a.m. and after 5p.m. on weekday, and all day of weekend. Because each ML has different amount of messages, we used ratio of messages, defined as amount of messages divided by total amount, of each ML group.

3. RESULTS AND INTERPRETATIONS

The ratio of messages in each hour is shown in Figure 2. The ML of hackers was active in the morning. On the contrary, ratio of committers' messages in the evening is comparatively higher than other ML groups. The ratio of messages in each day of a week is shown in Figure 3. Most developers work on weekday. The ratio of committers' messages on weekend is slightly higher than other ML groups.

Ratio of messages sent at overtime period in each year is shown in Figure 4. The ratio of messages of committers was increasing year by year while other ML groups did not show clear trends. This may suggest that many committers are recently required to work at overtime period by some reasons (e.g. too many patches to inspect). Ratio of messages sent on weekend by year is shown in Figure 5. In spite of the increase of committers' messages sent at overtime period, committers' messages sent on weekend did not increase. This may suggest that even if committers are willing to work at overtime period, they do not want to work on weekend.

4. CONCLUSIONS

We analyzed mailing lists archives of PostgreSQL. We focused on mail sent hour, days of a week of mail sent date, difference in hours and data among ML groups, and time trend. Our finding is that the ratio of committers' messages sent at overtime period was increasing year by year.

5. ACKNOWLEDGMENTS

This work is supported by the EASE (Empirical Approach to Software Engineering) project of the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

[1] http://hp.vector.co.jp/authors/VA024591/ (in Japanese)

Where is Bug Resolution Knowledge Stored?

Gerardo Canfora
Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
canfora@unisannio.it

Luigi Cerulo
Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
lcerulo@unisannio.it

ABSTRACT

ArgoUML uses both CVS and Bugzilla to keep track of bug-fixing activities since 1998. A common practice is to reference source code changes resolving a bug stored in Bugzilla by inserting the id number of the bug in the CVS commit notes. This relationship reveals useful to predict code entities impacted by a new bug report.

In this paper we analyze ArgoUML software repositories with a tool, we have implemented, showing what are Bugzilla fields that better predict such impact relationship, that is where knowledge about bug resolution is stored.

Categories and Subject Descriptors

H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement, Experimentation

Keywords

Mining Software Repositories, Impact Analysis

1. OVERVIEW

In [1] we introduced a method to predict the set of source files impacted by a new bug description submitted to a bugzilla repository. It takes advantage of the impact relationship extracted from CVS commit notes as suggested in [3]. In a set of four case studies, we obtained a top 1 precision ranging between 20% and 78%, and a top 30 recall ranging between 67% and 98%. The method builds, for each source file, a descriptor consisting of free text extracted from the set of fixed bugs and CVS commit notes that previously impacted it. An information retrieval algorithm scores each source file by measuring the similarity between its descriptors and the new bug description. The hypothesis is that the textual data carried by the bug tracking system during the bug fixing activity is a good descriptor of the impacted files to be considered in the impact analysis of future similar bugs. The similarity between descriptors is computed by using a probabilistic model that assumes that each term is associated with a topic, and that a document may be about the topic, or not [5]. The score of a source file descriptor d with respect to a bug is measured by using the following statistic measure about the term occurrences in source file descriptors:

$$S(d, bug) = \sum_{t \in bug} W_d(t)$$

where W is a weighting function directly proportional to the term frequency in the source file descriptor, and inversely proportional to the inverse document frequency [5].

A source file descriptor is represented with any combination of the following software repository fields: *notes*, the set of CVS commit notes; *short-descr*, the short bug description; *long-descr*, the long bug description; *comments*, the set of comments submitted by developers during bug resolution.

The model has been implemented in a tool, named Jimpa [2], that allows user to write a short explanation of a change and return the set of source files, ranked by their relevance with bug change description. The tool provides the support for setting information retrieval properties such as stop word list, stemmer algorithm, and software repositories fields to be included or excluded from the indexing process. Figure 1 shows a snapshot of the tool.

Data, provided by the tool, is stored in an intermediate database; we have used this database to perform the analysis presented in the following section that shows what are source file descriptor fields that contain more information about bug resolution.

2. MINING RESULTS

ArgoUML is an open-source UML modeling tool implemented in Java. Development started in 1998. The first bugzilla bug has been submitted in January 2000. Currently there are 2018 fixed bugs, 670 of which (about 33%) are referenced in CVS commit notes. The total number of Java source files is 1538. 6% of these files have a reference to more than 10 different bugs, while 40% of files do not have any bug reference.

We have performed a change impact prediction with the method introduced in [1] and with source file descriptors composed in different ways. In particular, we have used, as

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

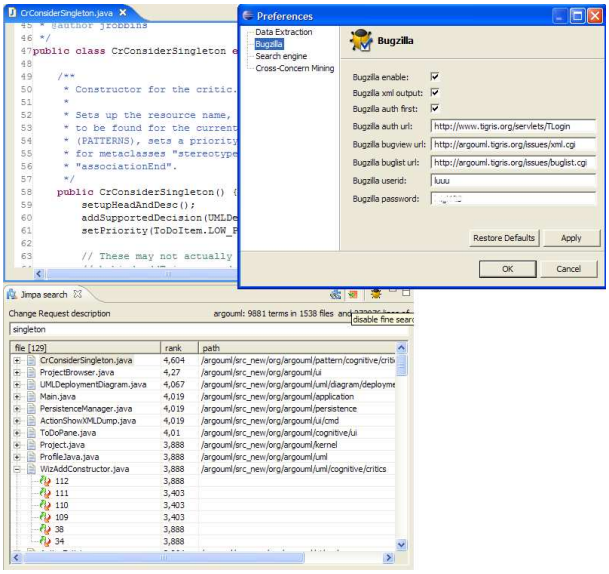


Figure 1: Tool snapshot

source file descriptor, each combination of bugzilla fields in order to put in evidence what is the field whose presence should give a better prediction performance.

Table 1 shows the top 1 precision and top 100 recall. The first is the percentage of cases in which the first retrieved source file is correct, while the second is the percentage of correct source files covered by the first 100 retrieved source files. Results show that the presence, in source file descriptor, of bug resolution comments give always the best performance. In particular, the overall best performance is obtained with a descriptor composed only with bug comments and CVS notes. This leads to consider that short and long descriptions submitted when the bug is discovered contain a partial knowledge about bug resolution, while most of the knowledge is contained in the comments submitted during the bug resolution process. On average, the presence of bug comments information gives an improvement of precision of about 5%.

Precision and recall have been computed using the leave-one-out assessment technique [4, 6] performed over 670 bugs, with short descriptions used as queries. For a given bugzilla bug we have predicted the set of impacted files by using an index without data regarding that bug. The predicted set of files has been then compared with the oracle set, that is the files impacted by that bug, recovered by considering the presence of the Buzilla id number in the revision comments of the files [3].

No evidence has been found for the dependence of prediction performance with other information retrieval parameters, such as, general English stop word list and Porter stemmer algorithm.

3. CONCLUDING REMARKS

Text mining of software repositories integrates information provided by source code analysis and gives new opportunities to support the software development process and to know new aspects about software evolution. It can be used not only for impact analysis but also, for example, to aggregate source files in a topic clusters.

Table 1: Performance dependencies

top 1 precision	top 100 recall	CVS notes	bug short decr	bug long decr	bug comments
0.232	0.791	×			×
0.212	0.802	×	×		×
0.191	0.795	×	×	×	×
0.163	0.792	×		×	×
0.161	0.790	×		×	
0.145	0.788	×	×	×	
0.126	0.754	×	×		
0.119	0.701	×			

Quality of text and project maturity are two factors that strongly impact every approach that takes advantage on free text stored in software repositories. Sometime CVS comments are used for communication rather that for description purpose and in almost all projects there is an initial period of transition that generates noise in both CVS and Bugzilla repositories. This leads to consider that results and issues obtained by applying data mining algorithms on software repositories can suggest new directions in developing more innovative configuration management and software development tools.

The open source community uses other repository for knowledge sharing, such as: mailing lists, newsgroups, and IRC conversations. They are rich of free text and it should be interesting to investigate how this information can be used in conjunction or as an alternative to CVS and Bugzilla.

4. REFERENCES

[1] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *METRICS '05: In Proceedings of the 11th IEEE International Software Metrics Symposium*, Como, Italy, 2005. IEEE Computer Society.

[2] G. Canfora and L. Cerulo. Jimpa: An eclipse plug-in for impact analysis. In *CSMR '06: In Proceedings of the 10th European Conference on Software Maintenance and Reengineering: Tools Demonstration*, Bari, Italy, 2006. IEEE Computer Society.

[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: In Proceedings of the 19th International Conference on Software Maintenance*, Amsterdam, Netherlands, 2003. IEEE Computer Society.

[4] K. Fogel and M. Bar. *Cross-Validatory Choice and Assessment of Statistical Predictions (with Discussion)*, volume 36. J. the Royal Statistical Soc., 1974.

[5] K. S. Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808, 2000.

[6] B. Ribeiro-neto and Baeza-yates. *Modern Information Retrieval*. Addison Wesley, 1999.

Mining Email Social Networks in Postgres

Christian Bird, Alex Gourley,
Prem Devanbu, Michael Gertz
Dept. of Computer Science, Kemper Hall,
University of California, Davis,
Davis, California Republic.
cabird,devanbu@ucdavis.edu

Anand Swaminathan
Graduate School of Management,
University of California, Davis,
Davis, California Republic.
aswaminathan@ucdavis.edu

ABSTRACT

Open Source Software (OSS) projects provide a unique opportunity to gather and analyze publicly available historical data. The Postgres SQL server, for example, has over seven years of recorded development and communication activity. We mined data from both the source code repository and the mailing list archives to examine the relationship between communication and development in Postgres. Along the way, we had to deal with the difficult challenge of resolving email aliases. We used a number of social network analysis measures and statistical techniques to analyze this data. We present our findings in this paper.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Empirical, Open Source*

General Terms

Human Factors, Measurement

Keywords

Open Source, Social Networks

1. INTRODUCTION

We have created a framework for mining publicly available OSS project data and using the results to answer questions about the activity in OSS projects. In an effort to test and validate our hypotheses based on earlier results from the Apache HTTP Server project, we have performed the same mining and analysis process on the Postgres SQL Server project[1]. We have mined source code repository activity and used mailing list archives to create a social network of developers and contributors to Postgres. We are hoping to answer the following questions:

- *Are the distributions of email activity, and the social network measures (such as in-degree and out-degree) similar in both projects?*
- *Is there a correlation between mailing list activity and development activity?*
- *Do the developers have significantly higher status than non-developers in the email social network?*

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

MSR'06, May 22–23, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

2. DATAMINING

The Postgres project is a stable and widely used piece of open source software with archives dating back to 1996. In order to mine social data from mailing list archives, we need various forms of information about each message sent on the list. Specifically, we need to know who sent a message, when the message was sent and if the message was sent in reply to a previous message. Mailing lists accomplish this "message linking" by assigning each message a unique message ID. Message a is a reply to message b if there is an *In-Reference-To* or *In-Reply-To* header in a's headers that has b's message ID in it. Unfortunately, although the mailing list archives for Postgres began in January of 1997, this method of using message ID's did not begin until January, 1998. We therefore restricted our mining effort to the time period from January, 1998 to February, 2006.

For the period in question, we found that there were 111,020 messages sent on the mailing lists (over 1,100 per month or 35 per day on average). We were able to parse 110,260 messages (approximately 99.3%). The remaining 760 messages were unparseable mostly due to malformed headers that lacked the *Message-ID* header crucial to our social network reconstruction. However, we believe that our results would not be significantly affected by the small proportion of unparseable messages

A serious hurdle to data collection was email aliasing. We found that during this time period, messages were sent to the list from 4,075 unique email addresses. Mailing list participants often use multiple email addresses, so for our analysis to be a valid, we need to remove the aliasing from the data. Each message sent on a mailing list has a name and an address of the sender. We have constructed an algorithm that uses a number of heuristics (such as address similarity, edit distance between names, etc.) and clustering to detect sets of email aliases that belong to one person. The results of this process are manually verified and editted for better results. Although it is not possible to completely remove aliasing based on name and address heuristics, (it's possible that the name, email pair ⟨shiby thomas, sthomas@cise.ufl.edu⟩ is the same person as ⟨david wetzel, dave@turbocat.de⟩, in which case our algorithm would miss it) we believe that our process is fairly accurate. Details of the aliasing algorithm are presented in the companion MSR paper[2]. After removing aliases we found 3,293 unique "identities" that we believe each correspond to one person. We used a similar technique in conjunction with online research (most OSS projects have a credits file or a developer info page[3]) to match CVS accounts to mailing list identities.

[2] http://wwwcsif.cs.ucdavis.edu/~bird/papers/msr06.pdf
[3] The email addresses of many Postgres developers can be found at http://www.postgresql.org/developer/bios

	changes	srcChanges	docChanges	outdegree	indegree	betweenness	mean	min	max
changes	1	0.974	0.936	0.768	0.782	0.765	3247	0	35883
srcChanges	0.974	1	0.885	0.769	0.785	0.769	2016	0	23345
docChanges	0.936	0.885	1	0.747	0.759	0.767	1231	0	12538
outdegree	0.768	0.769	0.747	1	0.992	0.948	0.0115	0	0.0679
indegree	0.782	0.785	0.759	0.992	1	0.956	0.0092	0.0001	0.0506
betweenness	0.765	0.769	0.767	0.948	0.956	1	.0246	0	0.2634

Figure 1: Cross-correlation table, (using Spearman's rank correlation) showing the relationship between the total number of changes, the changes to source, changes to documents, relative in-degree, relative out-degree, and betweenness. Average, min, and max are also shown. n=25

In addition to mining mailing list data, we also gathered data from the source code repository of Postgres (which uses CVS as its version control mechanism). During the period of interest, 26 CVS accounts were used. We were able to match email addresses to all but one of these. According to the developers[4], the *pgsql* account is used only to tag and package releases, and is not represented on the mailing list so we do not include it in our analysis. We tracked development by counting the number of changes to files over time and found 83,359 changes made to 4,108 files over the course of the time studied.

3. RESULTS

We constructed a social network based on the messages that were sent and replied to on the mailing lists. Three commonly accepted social network metrics were run on the resulting network on a per node basis; in-degree, out-degree, and betweeness. In general, developers had higher levels of all three metrics by at least an order of magnitude over non-developers. This indicates that developers hold positions of high status in the social network of contributors by multiple measures. A Student's t-test shows a significant statistical difference in the in-degree, out-degree and betweenness values for the population of developers and the population of non-developers. Figure 2 shows the social network of highly active Postgres mailing list participants (ties represent at least 150 messages between participants). The two most central participants, Bruce Momjian and Tom Lane, are also the most active CVS committers. The majority of the other participants in this network are also CVS committers. There are, however, nodes in this network that are not CVS committers and not all committers are in the network.

In addition, Figure 1 shows high levels of correlation between the social network measures and CVS activity. Similar to the results of our study of the Apache HTTP Server project, The social network metrics are higly correlated with source file changes. Unlike Apache, however, document file changes correlate to an equal degree. This may be due to the lower number of CVS developers (25 versus 78) and the fact that in this project, many developers work on both source code and documentation. Another possibility may be the number of document translations and how they are dealt with. We plan to mine other OSS projects to investigate this phenomenon futher.

We also examined the distribution of people with in-degree, outdegree, number of sent messages and number of replies. Consistent with data from the Apache project, each distribution exhibits a power-law character. This gives us confidence in our mining methodology and analysis as social processes tend to be characterized by power-laws.

[4]Marc Fournier and Tom Lane both explained this in responses to our inquiries regarding this account

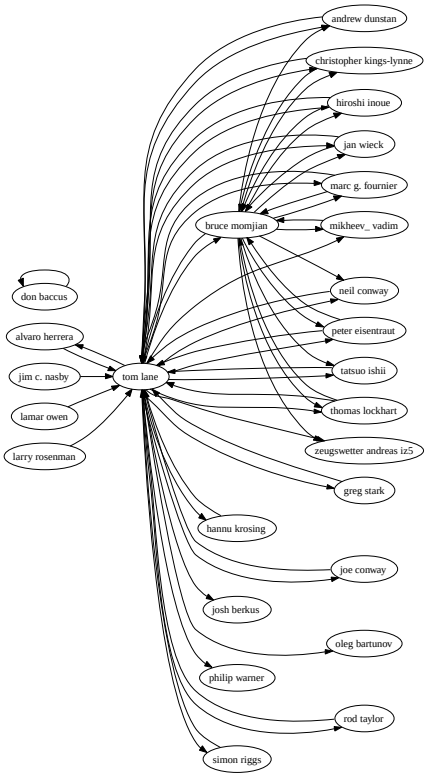


Figure 2: Social network of highly active Postgres mailing list participants

4. CONCLUSION

After mining and analyzing mailing list and source code repository data for the Postgres project we found that the distributions of email activity and social network measures were similar to those found in the Apache project. Our results indicate that developers hold higher levels of status in the social network than non-developers. We also found high correlations between various social network status metrics and source code development. This is consistent with our findings from the Apache project and gives us confidence in our hypotheses and methods. The discrepancy in correlation of document changes with social network status between projects indicates an area that requires further investigation

There is a significant body of related work, which is omitted from this summary for brevity. We refer the reader to our companion paper, "Mining Email Social Networks" accepted to MSR 2006 (located at http://wwwcsif.cs.ucdavis.edu/~bird/papers/msr06.pdf) for details.