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`

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.

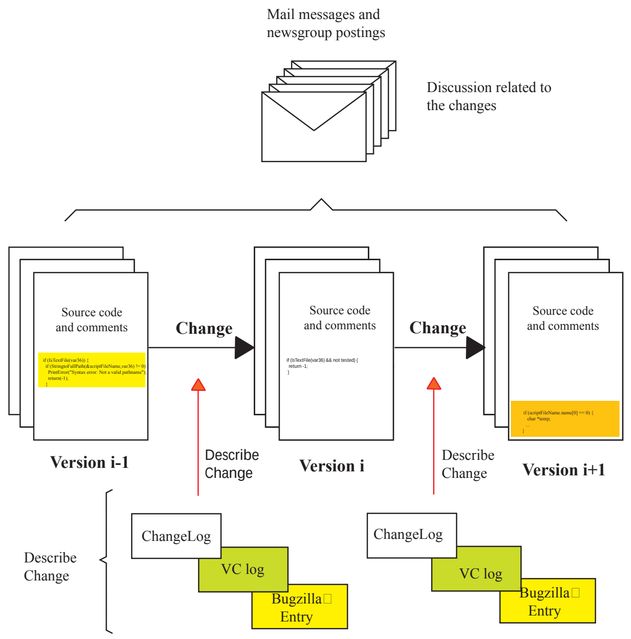


Figure 1: Evolutionary annotations are "horizontal" while source code comments are vertical.

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

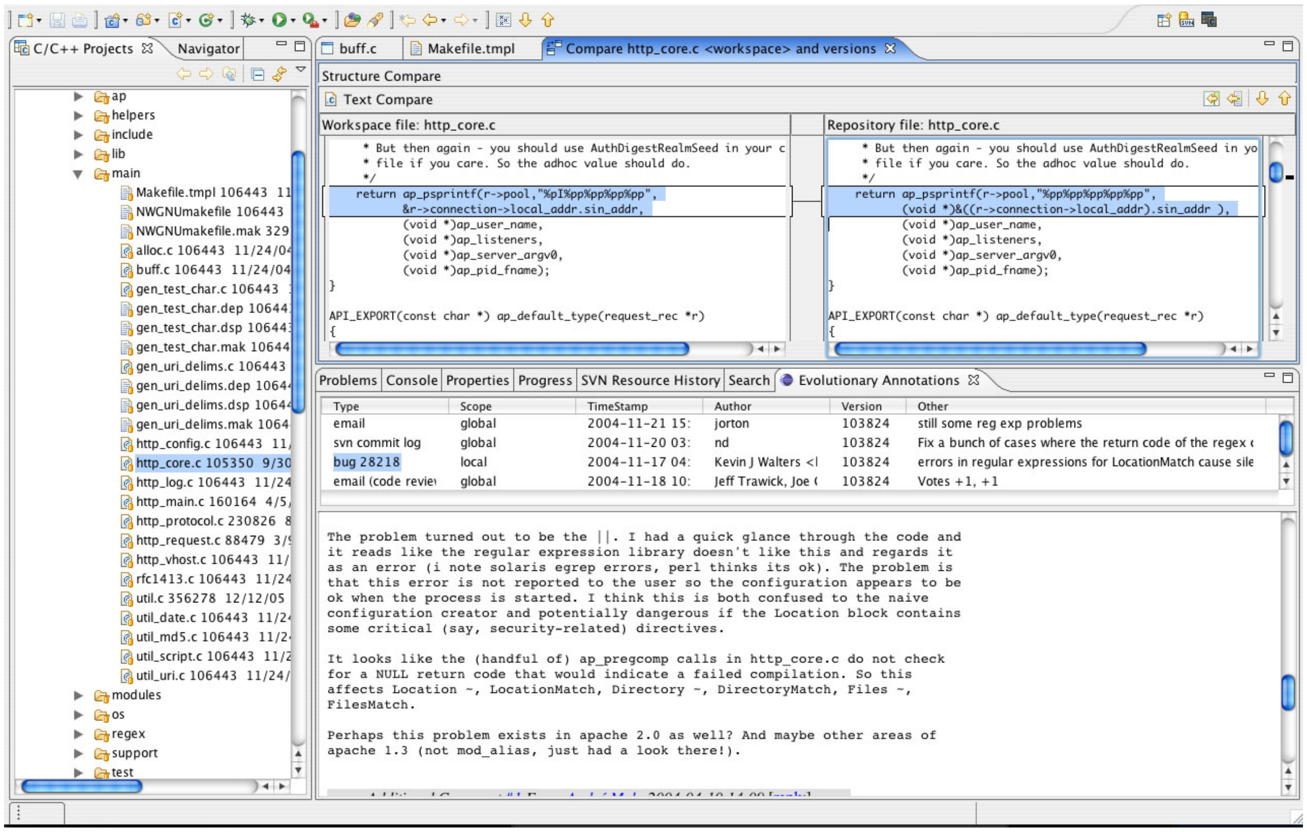
The proposed architecture consists of a database that contains all the evolutionary annotations that are correlated to the source code. As development artifacts are changed (e.g., source code changes are committed, email messages are sent, defects are reported, etc), the database is updated. A web service 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



Workspace file: http_core.c

```
     * But then again - you should use AuthDigestRealmSeed in your c
     * file if you care. So the adhoc value should do.
     */
    return ap_psprintf(r->pool,"%pI%pp%pp%pp%pp",
               &r->connection->local_addr.sin_addr,
               (void *)ap_user_name,
               (void *)ap_listeners,
               (void *)ap_server_argv0,
               (void *)ap_pid_fname);
}

API_EXPORT(const char *) ap_default_type(request_rec *r)
{
```

Repository file: http_core.c

```
     * But then again - you should use AuthDigestRealmSeed in yo
     * file if you care. So the adhoc value should do.
     */
    return ap_psprintf(r->pool,"%pp%pp%pp%pp%pp",
               (void *)&((r->connection->local_addr).sin_addr ),
               (void *)ap_user_name,
               (void *)ap_listeners,
               (void *)ap_server_argv0,
               (void *)ap_pid_fname);
}

API_EXPORT(const char *) ap_default_type(request_rec *r)
{
```

Type	Scope	TimeStamp	Author	Version	Other
email	global	2004-11-21 15:	jorton	103824	still some reg exp problems
svn commit log	global	2004-11-20 03:	nd	103824	Fix a bunch of cases where the return code of the regex
bug 28218	local	2004-11-17 04:	Kevin J Walters <l	103824	errors in regular expressions for LocationMatch cause sile
email (code reviei	global	2004-11-18 10:	Jeff Trawick, Joe (	103824	Votes +1, +1

```
The problem turned out to be the ||. I had a quick glance through the code and
it reads like the regular expression library doesn't like this and regards it
as an error (i note solaris egrep errors, perl thinks its ok). The problem is
that this error is not reported to the user so the configuration appears to be
ok when the process is started. I think this is both confused to the naive
configuration creator and potentially dangerous if the Location block contains
some critical (say, security-related) directives.

It looks like the (handful of) ap_pregcomp calls in http_core.c do not check
for a NULL return code that would indicate a failed compilation. So this
affects Location ~, LocationMatch, Directory ~, DirectoryMatch, Files ~,
FilesMatch.

Perhaps this problem exists in apache 2.0 as well? And maybe other areas of
apache 1.3 (not mod_alias, just had a look there!).
```

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.