An Empirical Approach to Software Archaeology*

Gregorio Robles, Jesus M. Gonzalez-Barahona, Israel Herraiz
GSyC, Universidad Rey Juan Carlos (Madrid, Spain)
{grex,jgb,herraiz}@gsyc.escet.urjc.es

Abstract

The term "software archaeology" provides a useful metaphor of the tasks that a software developer has to face when performing maintenance on large software projects. The source code of a program at any point in time is the result of many different changes performed in the past, usually by several people, which can be tracked when a version control system is used. We have designed a methodology for analyzing with detail the age of the source code in such cases, and have applied it to several large software projects. As a part of the methodology, we define a set of indexes which can help to characterize the history of a software system, and discuss how those could be used to estimate its past and future maintenance. We also show how our approach to software archaeology is simple both conceptually and computationally, but still very powerful at uncovering useful information.

Keywords: software archaeology, software maintenance, software evolution, empirical analysis

1. Introduction

The idea of applying the concept of archaeology[1] [1] to software maintenance can be tracked at least to the OOPSLA 2001 Workshop on Software Archeology. Software archaeology has been generally used for large old (legacy) systems, but it is valid for any type of software with independence of its age and size. While maintaining a given piece of software, developers have to understand source code that has usually changed many times in the past, producing a result which is the addition of all those changes. If the code is stored in a version control system, its complete history is available, and can be analyzed with appropriate tools. In this short paper, we will focus on the analysis of such a history from a macro point of view, gaining knowledge of the historical structure of a system as a whole, the same way that archaeologists gain knowledge of the history of an ancient city by studying what remains from the different constructions built in it.

For studying projects from this macro-archaeology point of view, we have designed a methodology, which is presented in this paper, and a set of tools to automate it. The methodology starts by determining, using information from the version control system, when and who modified for the last time each line of code. Then, the information for all lines is considered to calculate several indexes which provide useful information about the age of the code, the activity of developers in the past, the level of changes (maintenance), etc. Using this information we may also be able to estimate how much effort new changes would imply.

As case examples of the use of the proposed methodology we have selected nine libre (free, open source) software projects, most of which are among the hundred largest libre software applications included in the latest stable Debian GNU/Linux release[2].

The structure of this paper is as follows. The next section shows the methodology we propose for data extraction and analysis. After that, in section three, we apply our methodology and discuss the results obtained. The forth section introduces a set of indexes that we propose and briefly discuss. Finally, conclusions and further research goals are presented.

*This work has been funded in part by the European Commission, under the CALIBRE CA, IST program, contract number 004337, by the Universidad Rey Juan Carlos under project PPR-2004-42 and by the Spanish CICyT under project TIN2004-07296.

[1] In American English 'archeology'. The term comes from the Greek meaning '$\alpha\rho\chi\alpha\iota\omicron\varsigma$' (ancient) and '$\lambda\tilde{\iota}\gamma\omicron\varsigma$' (word/speech).

[2] Debian GNU/Linux is one of the most representative distributions, and probably the largest one. See details in `http://libresoft.urjc.es/debian-counting/sarge`

2. Methodology

To define the methodology, we have considered software projects which store source code in a version control system (in particular, CVS, although it could be easily extended to some other). CVS keeps record of every change in the code. It features a specific option ('annotate') which shows, for any line, the date and author of the last modification.

The process starts by obtaining, for every source file in the current snapshot of the software, the corresponding annotated files. They are stored and parsed. Source files are identified by applying certain heuristics on the file names (for instance, those ending in .c are supposed to be C source files). For considering just code, blank lines and comments are removed also using some other heuristics. In addition, we run some error-correction routines which check for common errors found when mining data from CVS; in order to verify our heuristics, we have compared the number of SLOCs obtained with SLOCCount[3] with the number of lines obtained after applying our heuristics.

Once the annotated files have been parsed, and the mentioned heuristics applied, the resulting data is normalized and inserted into a database, which will be later queried for getting statistical information. This process is performed by a set of scripts which are also responsible for the generation of the kind of graphs shown in this short paper.

3. Case studies

We have applied the described methodology to the code produced by nine libre software projects. They show a great variety from many points of view (age, size, complexity, number of developers, etc.), but all of them are included in major GNU/Linux distributions, which is an evidence of their popularity. In total, our case studies sum up to 9.5 millions lines of code, written mainly in C and C++, and 52,975 source code files. Table 1 presents the most important facts about the code considered.

3.1. Remaining lines

Figure 1 shows how many lines remain untouched since any past date for all the projects relative to the size of each project. The horizontal axis is time, while the vertical axis is measured in percentages (being 100% the current size of the project). In the figure we can see what which fraction of code is newer than a date. For example, for the case of Apache, approximately 60is posterior to December 1998.

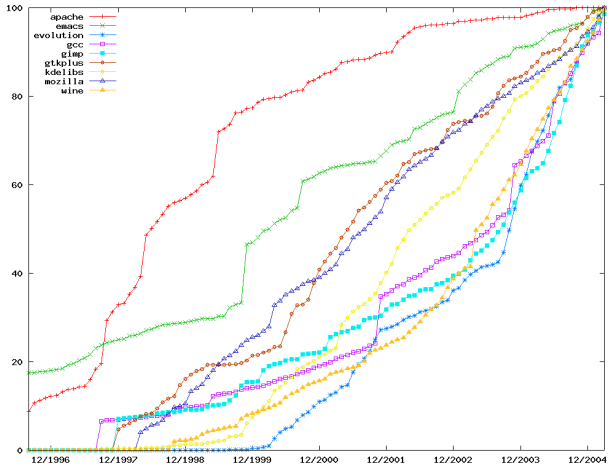

Figure 1. Remaining lines (relative values)

Interestingly enough, the code in all projects is young. Besides Apache 1.3, at least half of the code in all of them is younger than 5 years. Even the code base for Emacs, which we had selected as a legacy system, has a large fraction (up to 70%) which is less than 7 years old.

Apache 1.3 has to be considered separately, since developers are now focused on Apache 2.0, where the main development effort is taking place. However, we expected that at least some corrective maintenance effort would be happening in 1.3, but at least since 2003 that does not seem to be the case.

In the other end of the spectrum, with most of the code being really new, we find GCC, Evolution, GIMP and Wine. in all these cases, this is due, probably, to recent refactorings of the code, including structural and organizational changes.

4. Indexes

To get useful information from software archaeology, it is convenient to use some parameters that help to characterize the history of the project from this point of view. This is the reason why we have defined some indexes that may help to infer some properties of the corresponding development and maintenance process.

[3] We use the '–duplicates' option which counts duplicated files twice as our tools, contrary to SLOCCount, do not filter them out. SLOCCount is available at `http://www.dwheeler.com/sloccount`

Project	Start	Vers. 1.0	Oldest line	SLOCs	SLOCCount	Percent.	Files	Authors
Emacs	(1976)	1985	May 85	974,407	991,552	98.3%	1,522	136
GCC	1985	1987	Sep 97	2,191,764	2,262,632	96.9%	22,349	218
Wine	1993	-	Oct 98	1,033,318	984,710	104.9%	2,201	2
GTK+	1994	Apr 98	(Dec 97)	387,413	389,723	99.4%	839	114
The GIMP	1994	Jun 98	(Dec 97)	548,410	552,473	99.3%	2,244	71
Apache 1.3	1995	Jun 98	Feb 96	82,909	85,758	96.7%	269	51
kdelibs	1997	Jul 98	May 97	605,528	613,742	98.6%	3,131	363
Evolution	1998	Dec 01	May 98	205,278	207,069	99.1%	816	79
Mozilla	(1998)	Jun 02	(Apr 98)	3,414,387	3,510,691	97.3%	19,604	567

Table 1. Summary of the case studies. Columns contain the project name, the year the project started its development, the date of its release 1.0, the number of SLOCs according to our methodology, the number of SLOCs according to SLOCCount, the coincidence for both figures, the number of files, and the authors identified in the current version.

4.1. Definition of the indexes

- **Aging** (measured in SLOC-month). It is a direct measure of how much the software is aging.

$$Aging = \sum_{n=1}^{N-1} lines_n \tag{1}$$

  where n is the month number, being n=1 the first month of the project and N the current one. Notice that the last month is not taken into account.

This index is defined after Parnas' well-known software aging [2] concept, although we only have in mind one of the factors. If we would stick to Parnas' original definition of aging, then we should take into account changes performed on the system, and not only that the software gets old as humans do.

- **Relative aging**. This index makes it possible to compare the *aging* for several projects. It is measured in months and can be obtained from following equation:

$$RelativeAging = \frac{Aging}{lines_N} \tag{2}$$

  where N is the last month considered.

Relative aging represents the amount of time necessary to have the same aging, had the project started with the current number of lines. Of course, it can also be understood as the number of months needed to double the current *a*ging of the project if the system is not touched anymore.

- **Relative 5-year Aging**: relative size to itself as if the project were 5 years old.

$$Rel5yA = \frac{Aging}{60 \cdot lines_N} \tag{3}$$

  where N is the last considered month

Relative 5-year aging allows for easier comparison, defining 5 years as the moment for a system to become 'old'. It is also a needed step for defining the *absolute 5-year aging* index (which will be presented later).

- **Progeria**[4]. As *relative aging* measures the amount of time needed to double the *aging* value, we can compare it to the amount of time needed to double the code base.

$$Progeria = \frac{RelativeAging}{50\% of CurrentCode} \tag{4}$$

Values of progeria lower than 1 are indicative of active maintenance. Projects featuring those indexes have not to fear the consequences of high values of *aging*. However, values above 1 imply that *aging* is growing faster than software maintenance activity and therefore are prone to showing more and more problems.

A new index that provides a value relative to a fixed-size and a fixed-time software system will enable comparison among projects.

- **Absolute 5-year aging**: relative size as if the project had 100 KSLOC and had been started 5

[4] Progeria is a genetic condition which causes physical changes that resemble greatly accelerated aging in sufferers. Source: WikiPedia

Project	Size	Age	Aging	Rel. Aging	Rel5yA	Progeria	Abs5yA
Emacs	974,043	239	62,419,261	64.1	1.07	0.93	10.40
GCC	2,188,033	91	65,558,122	30.0	0.50	0.65	10.93
Wine	1,028,820	78	26,926,319	26.2	0.44	0.80	4.49
GTK+	387,333	88	16,938,898	43.7	0.73	1.04	2.82
The GIMP	540,540	98	16,002,332	29.6	0.49	0.59	2.67
Apache 1.3	82,909	110	6,161,847	74.3	1.24	1.10	1.03
kdelibs	604,888	95	20,089,807	33.2	0.55	1.04	3.35
Evolution	204,951	99	4,796,800	23.4	0.39	0.66	0.79
Mozilla	3,786,735	84	161,394,929	42.6	0.71	1.00	26.90

Table 2. Archaeology indexes for our case studies. Size is given in SLOC, Age in months, Aging in SLOC-month, Relative Aging in months, Progeria, Rel5yA and Abs5yA are indexes.

years (60 months) ago. Serves for comparison purposes among projects.

$$Abs5yA = \frac{Aging}{60 \cdot 100K} \quad (5)$$

where N is the last considered Month.

4.2. Application to the case studies

Table 2 shows how the aging index is not too useful for comparison purposes (although it provides a good idea of the absolute aging). However, relative aging allows for those comparisons. We can see in the corresponding column of the table a summary of the information in figure 1. Apache and Emacs are the systems with the highest relative aging. Evolution, Wine and The GIMP have values in the 20s, which mean that they are still in actively maintained.

With respect to progeria, it can be said that it shows how Mozilla balances aging and evolution, while there are four projects which are becoming old systems: Apache and Emacs (which at this stage of the analysis is not surprising at all), but also GTK+ and kdelibs.

The absolute 5-year aging depends on the size, and has been presented as a proxy of maintainability. It shows that Apache, even having high progeria and aging is still more *friendly* to be maintained than the rest of systems (except for Evolution) because of its small size. Emacs and GCC, even having the latter two times the size of the former, have similar values, while GTK+ and GIMP also show this behaviour.

5. Conclusions and further research

In this paper we have presented an empirical application of the archaeology concept to the macro study of projects maintained in version control systems, with special focus on libre software projects. We have devised a methodology for that study, from which we have defined several indexes which can be used to summarize the development process from the point of view of aging and maintenance.

One of the key findings of this work has been to show that the application of the methodology to the case examples has provided some insight about the maintenance efforts, and the maintainability of the corresponding projects. From a more general point of view, the characterization of a project by several indexes that contribute with useful information about its age and maintainability is probably the key contribution of our work and may help in the decision-taking process by the development teams in libre software projects or by the management team in industrial software companies.

There are many possible future lines of research to explore this approach. First of all, we are looking for better ways of visualization of the archaeological results from a macroscopic point of view. We are also interested in finding relationships with the parameters used in software evolution studies, and in correlating them with effort estimation.

As a summary, we believe that software archaeology provides an interesting framework for digging in the past of a project, so that we can learn patterns and information relevant to infer its future.

References

[1] A. Hunt and D. Thomas. Software Archaeology. *IEEE Software*, 19(2):20–22, 2002.

[2] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, Sorrento, Italy, May 1994.