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 in uncovering useful information.

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

1. Introduction

Although the concept of software archaeology[1] [12] is not new neither unknown to the software engineering community, there have been very few studies from this point of view, and up to now archaeological empirical analysis have been very rare.

The idea of applying the concept of archaeology to software maintenance can be tracked at least to the OOPSLA 2001 Workshop on Software Archeology which was organized by Ward Cunningham et al. The first of the assumptions proposed for the workshop is in fact the rationale for using the archaeology concept in software comprehension:

> "[Software] [a]rch[a]eology is a useful metaphor: programmers try to understand what was in the minds of other developers using only the artifacts left behind. They are hampered because the artifacts were not created to communicate to the future, because only part of what was originally created has been preserved, and because relics from different eras are intermingled."[2]

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 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.

In the case of libre (free, open source) software[3] this kind of knowledge can be specially important, since due to the usual lack of design [24], maintenance is an activity even more important than usual. In this respect, the age of the lines to be changed for a given maintenance activity is important. For instance, fragments

*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 'αρχαιος' (ancient) and 'λόγος' (word/speech).

[2] http://www.visibleworkings.com/archeology/

[3] 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 FSF) or "open source software" (according to the OSI).

of code which are too 'old' (remain in the system from early stages in the development process) are less maintainable, since the developers that coded them may not be part of the team anymore [9]. Therefore, in this case knowing the amount of code which was modified for the last time by developers that are no longer active in the project will be important to estimate the degree of difficulty of the maintenance process. And this is just an example of the kind of useful information that archaeology can provide.

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 software projects, most of which among the hundred largest libre software applications included in the latest stable Debian GNU/Linux release[4].

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 discuss. Finally, before the conclusions we have inserted two sections in which we present related work, discuss the usefulness of our approach and suggest further research directions that should be followed in the future. As case studies we have selected nine libre software projects which are described briefly in an appendix.

2. Methodology

To define the methodology, we have considered software projects that store source code in a version control system (in particular, CVS, although it could be easily extended to some other). Fortunately, there are a lot of libre and non-libre software projects in this situation: more than 10,000 projects hosted at SourceForge in 2003 [11], or the 11 largest projects in Debian 2.2 [8] satisfy this condition (all of them, except for Linux, use CVS).

CVS stores information about every change in the code. It features a specific option ('annotate') which shows, for any line, the date and author of the last modification. As an illustrative example, consider the (slightly modified) excerpt of the information obtained for the src/keymap.c file from Emacs:

```
[...]
1.33 (A 19-May-93):/* A char with (...)
1.33 (A 19-May-93):   in a string (...)
1.33 (A 19-May-93):   character. */
1.1  (A 06-May-91):extern Lisp_Object m;
1.1  (A 06-May-91):
1.55 (B 16-Jan-94):extern Lisp_Object V;
1.55 (B 16-Jan-94):
1.209(C 25-Oct-00):/* Hash table (..) */
1.209(C 25-Oct-00):static Lisp_Object w;
[...]
```

The structure of this output (which will be called 'annotated file') is as follows. The first column shows the file revision in which the current content of this line was introduced. The string in parenthesis corresponds to the username (in this example, A, B, C) and the date of the last commit. Finally, the current content of the line is found.

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. For instance, some lines show dates which do not correspond to reality, probably due to a temporary misconfiguration of the server clock (this happened in 8 lines from Evolution and 235 from The GIMP). In those cases, we interpolated the dates considering those of the previous and the next versions. Another case is related to branches, which are considered as having the next major version number. This happened, for instance, for 37 lines (affecting 9 files) in the Mozilla project, with a total of about 3.5 millions of lines.

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

[4]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`

is performed by a set of scripts which are also responsible for the generation of the kind of graphs shown in this paper.

The whole process, from source code retrieval to statistical analysis and graph generation, can be automated. We have built a tool which groups all the scripts for this task, DrJones, which is available as libre software[5]. This way other research groups have the possibility to use and enhance it.

2.1. Discussion and limitations of the methodology

The approach we have presented has the limitation of being only applicable to software projects which provide over a versioning system. If the project to analyze does not make use of one, but we have the historic versions of it (and the dates of their release), there exists the possibility of building a versioning system from them. This process can be automatized, although some information from the original analysis will be missing as for instance the real date of the modification of lines (which will be set to the one where the release took place) or information regarding authorship.

Other problems arise because of certain characteristics (or lack of them) in the versioning system. For instance, in CVS there exists no way for the CVS client to move files from one directory to another, so users have to delete and create a file again in the new location which actually makes our software archaeology approach error-prone. Newer versioning systems, such as Subversion, do not have this problem.

Finally, a third limitation is the insertion of external code, where vertical lines appear for some projects (as we will see for some of the case studies). This inserts errors in the analysis regarding the date and authorship. On the other hand, although the starting date of this code is not accurate it is code that has to be maintained, so this limitation should not be that problematic.

We have verified our heuristics for filtering lines of code by computing the source lines of code with David Wheeler's SLOCCount[6], a widely tested tool that has been used in previous studies [8, 10]. The figures produced by SLOCCount have been compared with the number of lines obtained with our procedure, showing low enough differences to consider our approach sufficient (see table 1 for details).

[5] http://libresoft.urjc.es/index.php?menu=Tools&Tools=DrJones

[6] 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`

3. Case studies

We have applied the described methodology to nine projects from the libre software world (the rationale for this selection is discussed in appendix A). 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 analyzed source code files. Table 1 presents the most important facts about the code considered.

The results of applying the methodology are presented in the following subsection.

3.1. Remaining lines

Figure 1 shows how many lines remain untouched since any past date for all the projects. Time is represented in the horizontal axis, while the remaining lines for any given date are found in the vertical one. It can also be understood as a curve showing, for each date, the number of lines that are older than it. Hence, periods without maintenance activity (i.e. projects with no changes) would show an horizontal line. This is almost the case for Apache 1.3 (at the bottom of the figure), but not for the rest of the projects which show linear (as for instance Mozilla) or super-linear (for instance Emacs) trends.

Notice, in any case, that super-linearity in this figure does not mean that the growth of the project is also super-linear: the number of remaining lines is the sum of changed and aggregated lines.

Figure 2 shows the same information, but now the curves are relative to the size of each project. The horizontal axis is still time, while the vertical axis now is measured in percentages (being 100% the current size of the project). In it we can read, for each point in time, the fraction of code that remains from the past.

Interestingly enough, the code in all projects is notoriously young. Besides Apache 1.3, at least half of the code in all of them is younger than 5 years, as can be seen with more detail in table 2 (where we show, as a complement to figure 2, when 30%, 50% and 80% of the current code is present). 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

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.

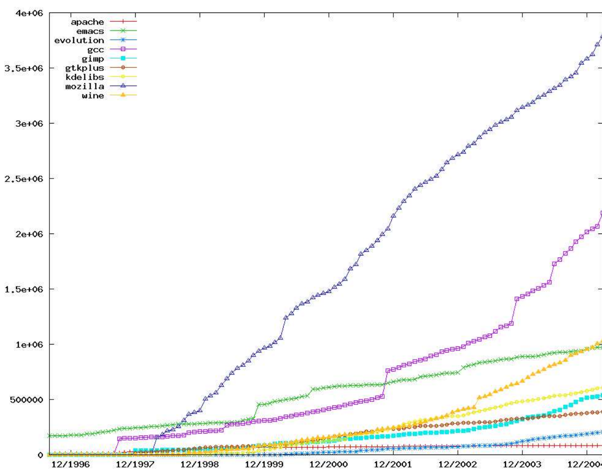

Figure 1. Remaining lines (absolute values)

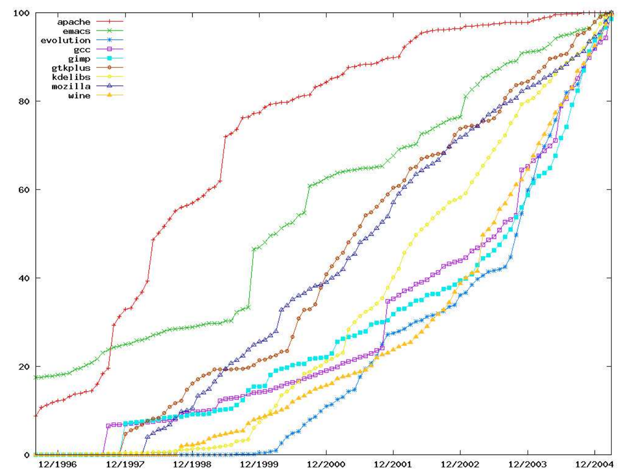

Figure 2. Remaining lines (relative values)

would be happening in 1.3, even if adaptative or perfective maintenance [27] is not performed. 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.

From another point of view, these graphs may be seen as estimators for future maintenance effort. Software with a high amount of 'old' lines of code is more difficult to maintain as authors may forget the rational of the changes or they may even be not more part of the project. In the next subsection we will study this with some more detail.

3.2. Remaining contributions from authors

Figure 3 shows the percentage of lines authored by developers not part of the project anymore. We will call those lines 'orphaned lines', because in some sense their author seems to be no longer present. For instance, projects that do not lose their developers will have little orphaned lines for the whole duration of the development, and therefore will show almost shapes close to horizontal. On the other hand, projects which have suffered a high loss of authors will have steep curves.

This figure has a clear relationship with the availability of knowledge within the project. Those projects losing developers will also experiment the loss of their ex-

Project	30%	50%	80%
Emacs	Jun 99 (69)	Apr 00 (59)	Jan 03 (26)
GCC	Nov 01 (40)	Jul 03 (19)	Jul 04 (8)
Wine	Jul 02 (32)	May 03 (21)	Jul 04 (8)
GTK+	Jul 00 (56)	Jun 01 (45)	Aug 03 (19)
The GIMP	Nov 01 (40)	Sep 03 (17)	Sep 04 (6)
Apache 1.3	Nov 97 (88)	Jun 98 (81)	Jun 00 (57)
kdelibs	May 01 (46)	May 02 (34)	Dec 03 (14)
Evolution	Apr 02 (35)	Nov 03 (15)	Jul 04 (8)
Mozilla	Apr 00 (59)	Sep 01 (42)	Sep 03 (17)

Table 2. Date when 30%, 50% and 80% of the code in March 2005 is already present. Numbers in brackets are months passed.

pertise about the system, which will have an impact on future maintenance efforts. It is not the same to maintain and enhance a system having the original authors in the team than having newcomers which have to embrace a software comprehension process before they become productive.

The curves in this figure have many steps since they are related to the whole amount of lines contributed by 'active' developers at any time. Hence, when a developer leaves the project we see a vertical line, longer or shorter according to his past contributions. It is remarkably, for instance, the case for Emacs in November 2003, when Richard Stallman (the original author) contributed his last change.

It can also be shown how values in this figure have to be below those in figure 2, since the remaining lines for a developer are dated as of his last contribution, and not as of their own date (which has to be in any case previous to the last contribution).

Table 3 helps in the understanding of figure 3 by presenting the 30% and 50% values of orphaned lines. .

4. Indexes

We have shown so far that an empirical approach to software archaeology uncovers a good deal of data. To get useful information from it, 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. This proposal goes in the same direction than the Maintainability Index proposed by Oman et al. [16], which has been applied recently by Samoladas et al. to libre software [23].

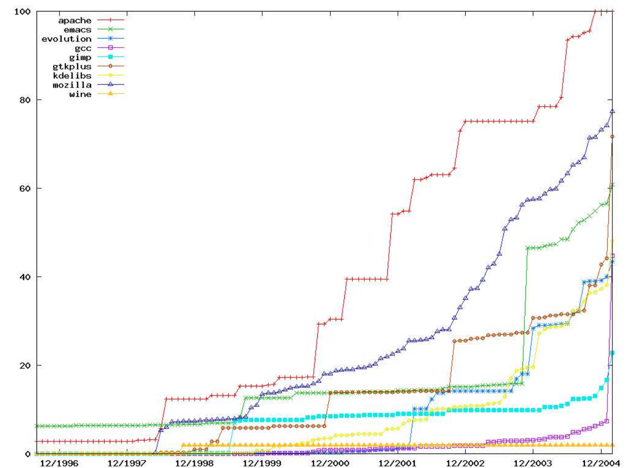

Figure 3. Orphaned lines in time (relative)

Project	30%	50%
Emacs	Nov 03 (16)	Jul 04 (8)
GCC	Feb 05 (1)	Mar 05 (0)
Wine	Mar 05 (0)	Mar 05 (0)
GTK+	Jan 04 (13)	Feb 05 (1)
The GIMP	Mar 05 (0)	Mar 05 (0)
Apache 1.3	Nov 97 (88)	Jun 98 (81)
kdelibs	Jul 04 (8)	Mar 05 (0)
Evolution	Jul 04 (8)	Mar 05 (0)
Mozilla	Oct 02 (29)	Jul 03 (20)

Table 3. Date when 30% and 50% of the code has an author which will make no more changes until March 2005. Numbers in brackets are months passed.

4.1. Definition of the indexes

- **Aging** (measured in SLOC-month). It is a direct measure of how much the software is aging. It can be calculated as the area under the curve in figure 1.

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

  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 [17] 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. We lack this information in archaeological studies, but other methods could be implemented to extract the required data from a versioning system and end up with a metric that would fit the original definition more accurately.

In any case, our *aging* index gives an idea of how 'old' a software system has become, but it does not provide much information about how much it has been maintained, neither about how easy it will be to maintain it in the future. In addition, the SLOC-month figures are difficult to be compared from project to project, as it can be seen in table 4. This leads us to the definition of *relative aging*.

- **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:

$$RelativeAging = \frac{Aging}{lines_N} \quad (2)$$

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.

$$Rel5yA = \frac{Aging}{60 \cdot lines_N} \quad (3)$$

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**[7]. 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.

$$Progeria = \frac{RelativeAging}{50\% of CurrentCode} \quad (4)$$

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

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 more than software maintenance activity and therefore are prone to showing more and more problems.

So far we have only considered *aging* to find out how old the system is, or how old it will get. But we have not found a value that shows how maintainable the system is. The fact that for previous indexes we have considered factors relative to the software size itself can be confusing, as systems that are by far larger may have values which make them be 'younger' than much smaller systems and thus easier to maintain. This means that those indexes cannot be taken as estimators of the effort by development teams taking over the software maintenance process.

Therefore, we propose a new index that provides a value relative to a fixed-size and a fixed-time software system. This will enable comparison among projects.

- **Absolute 5-year aging**: relative size as if the project had 100 KSLOC and had been started 5 years (60 monthts) ago. Serves for comparison purposes among projects.

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

where N is the last considered Month.

So far, we have considered only indexes related to source code, but software archaeology provides also information related to the authors. Indexes up to the moment did not consider changes in the development team. There is some research considering those changes, such as the quantification of the half-life [22] of the *core* team for some libre software projects (defined as the time required for a certain group of contributors to fall to half of its initial population).

One of the limitations of the following indexes is that the curves from which they are inferred are, contrary to those shown previously, not continuous with time, which may imply some undesired behaviours.

In the line of the *aging* index we can infer from figure 3 a similar index for those lines for which their author already left the project.

- **Orphaning** (in lines-month): gives the amount of lines that prevail from developers that are not active in the project anymore, multiplied by the number of months of inactivity.

$$Orphaning = \sum_{n=1}^{N-1} lines_n \quad (6)$$

where n is the month number, being n=1 the first month of the project and N the current one.

As for the *aging* values, the *orphaning* values are difficult to compare from project to project. We could calculate relative indexes as we have done for source code lines above, but since they depend on the same values, having a *factor* should be sufficient for doing transformations.

- **Orphaning factor**: the number of lines-month given by the *o*rphaning index is by definition equal or smaller than the *a*ging of the system. It can be seen as the fraction of old code wrote by developers not participating in the project anymore.

$$OrphaningFactor = \frac{Orphaning}{Aging} \cdot 100 \quad (7)$$

We multiply it by 100 in order to have it in percentages.

The orphaning factor may be seen as an index that shows how much of the existing code is supported by its original authors. Low values mean that the development / maintenance team is still available in the project. This ensures continuity and is indicative of lower maintenance costs (compared to the situation where it is to be maintained by newcomers).

The orphaning factor should be used with care. A first impulse would be to multiply it by the absolute 5-year aging (Abs5yA) in order to obtain an index that combines maintainability and current expertise of the team. This would be misleading, throwing for instance GCC a similar value as to Apache. The fallacy is due to the fact that by doing this we assume that authors dispose over an unlimited memory and remind all additions and changes despite the time that has passed. This, of course, cannot be considered for large systems and for long development/maintenance periods.

4.2. Application to the case studies

Once the indexes have been defined, we can study them in our case examples (see table 4), and comment some particular cases.

From this table it can be seen how the aging index is not quite useful for comparison purposes (although it gives 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 2. 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 (that's why it appears in Figure 1 as a linear function), 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.

While the orphaning index helps little in comparing projects, the orphaning factor provides very useful information. It shows small values for GCC and Wine, very high values for Apache and Mozilla, and medium values (around 25% to 35%) for the rest of systems). This may give an idea of how much experience the current team has with the system.

5. Related work and discussion

The use of historical data from software development is not a new topic for the software engineering community. But most of the approaches that have used such data have focused on evolutionary aspects [3, 15], or on looking at the reasons for the most common behaviours found in a software development project [15]. Of course, availability of the source code as well as public access to software tools used during the development process in libre software projects have made them gain attention recently, and some methods have already been proposed to automate these processes [5, 21].

In addition, evidence has been found [10] that the amount of available libre software, measured in source lines of code, gets doubled around every two years. This means that the code base to be maintained grows steadily in time and that bigger efforts have to be taken into consideration.

In this context, we have presented a brief introduction to why software archaeology may be useful in relation to software maintenance and software evolution. Further research has to be performed in order to see what other information can be obtained from such an analysis and how it can be connected to 'classical' analysis.

Specially interesting are the limitations of the current approach for software archaeology, and how we can con-

Project	Size	Age	Aging	Rel. Aging	Rel5yA	Progeria	Abs5yA	Orphaning	Orph Factor
Emacs	974,043	239	62,419,261	64.1	1.07	0.93	10.40	18,585,711	29.78
GCC	2,188,033	91	65,558,122	30.0	0.50	0.65	10.93	3,427,811	5.23
Wine	1,028,820	78	26,926,319	26.2	0.44	0.80	4.49	1,521,212	5.65
GTK+	387,333	88	16,938,898	43.7	0.73	1.04	2.82	5,218,899	30.81
The GIMP	540,540	98	16,002,332	29.6	0.49	0.59	2.67	3,595,296	22.47
Apache 1.3	82,909	110	6,161,847	74.3	1.24	1.10	1.03	3,290,623	53.40
kdelibs	604,888	95	20,089,807	33.2	0.55	1.04	3.35	5,023,152	25.00
Evolution	204,951	99	4,796,800	23.4	0.39	0.66	0.79	1,665,455	34.72
Mozilla	3,786,735	84	161,394,929	42.6	0.71	1.00	26.90	90,902,668	56.32

Table 4. Archaeology indexes for our case studies. Size is given in SLOC, Age in months, Aging and Orphaning in SLOC-month, Relative Aging in months, Progeria, Rel5yA and Abs5yA are indexes and the Orphaning factor is given in percentage.

nect it with the more mature view of software evolution. Also, we foresee promising lines related to cost estimation, and present some final remarks about some information that we have gathered but not analyzed yet.

5.1. Relationship with software evolution

The laws of software evolution [13] have some relationship to the archaeology concept: the former considers the total history of the project starting from the first time the software is publicly released, while the latter is only interested in what remains from the past. Godfrey et al. [7] studied, from this point of view, a libre software project (Linux) and found that it did not follow at least one of the software evolution laws, as its growth is super-linear. But evolution is only one side of the coin, and maintainability has also to be taken into account. In this sense, a study by Schach et al. [25] looked at the maintainability of the Linux kernel over several versions in time, throwing as a result that the dependencies of Linux may make it difficult to maintain in the near future.

Other works that have applied similar ideas come from the re-engineering area. For instance, the Yesterday's Weather [6] method looks at the most recent changes applied to a system, considering that recently changed parts are those more likely to change in the near future [26]. In comparison to software archaeology, this is like taking into account only the last months of development.

Further research should be performed in order to see how to relate both views. Does any kind of transformation exist between software evolution and software archaeology? What additional information is necessary for such a transformation?

5.2. Cost estimation

Ramil et al. [20, 19] have already performed some research on effort estimation based on change records of evolving software. Software archaeology and the indexes that we have presented so far may also be used as a starting point for the estimation of future software maintenance and evolution costs. The key ideas for walking that path are:

- A piece of software on which an author worked recently is (for him) easier to maintain than other in which he worked long ago.
- Maintaining software with which an author has never dealt is more difficult than if he has already worked on it.
- Having the possibility of asking the original author of some code to be changed makes life easier than when that developer is not available anymore.

6. 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 attention to 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. We have also built some tools that automate the analysis, and applied them to nine carefully selected libre software projects. To finish, we discussed some limitations and needed complementary research lines.

One of the key findings of this work has been to show that the application of the methodology to the case examples has provided much insight about the maintenance efforts, and the maintainability of the corresponding projects. Finding, for instance, that projects considered as legacy (such as Emacs) has around 70% of the code younger than 7 years, or that Apache 1.3 is seldom maintained even being the most used WWW server worldwide nowadays.

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 macro 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. In a more deep line, we are interested in characterizing the continuous release process, common in libre software projects, identifying its developments and maintenance parts, and relating them to other factors.

Some other quantitative approaches could also lead to interesting results from a software archaeology point of view. Among them, studying how the changes in documentation and in-line comments affect the maintenance process, and even simply repeat the same analysis presented in this paper for comments in code.

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] R. Ferenc, I. Siket, and T. Gyimóthy. Extracting facts from open source software. In *Proceedings of the International Conference in Software Maintenance*, pages 60–69, Chicago, IL, USA, 2004.

[2] 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, The Netherlands, September 2003.

[3] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 160–170, 1997.

[4] D. German. An empirical study of fine-grained software modifications. In *Proceedings of the International Conference in Software Maintenance*, Chicago, IL, USA, 2004.

[5] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, 2003.

[6] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *20th International Conference on Software Maintenance*, pages 40–49, September 2004.

[7] 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.

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

[9] J. M. Gonzalez-Barahona and G. Robles. Unmounting the "code gods" assumption. Technical report, 2003.

[10] J. M. González-Barahona, G. Robles, M. Ortuño Pérez, L. Rodero-Merino, J. Centeno González, V. Matellan-Olivera, E. Castro-Barbero, and P. de-las Heras-Quirós. Analyzing the anatomy of GNU/Linux distributions: methodology and case studies (Red Hat and Debian). In S. Koch, editor, *Free/Open Source Software Development*, pages 27–58. Idea Group Publishing, Hershey, PA, USA, 2004.

[11] K. Healy and A. Schussman. The ecology of open-source software development. Technical report, University of Arizona, USA, Jan. 2003.

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

[13] M. Lehman, J. Ramil, P. Wernick, and D. Perry. Metrics and laws of software evolution - the nineties view. In *Proceedings of the Fourth International Software Metrics Symposium*, Portland, Oregon, 1997.

[14] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of Open Source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[15] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, pages 120–130, October 2000.

[16] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *International Conference on Software Maintenance*, pages 337–344, Los Alamitos, CA, 1992.

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

[18] J. W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software

products. *Transactions on Software Engineering*, 30(4), April 2004.

[19] J. F. Ramil and M. M. Lehman. Effort estimation from change records of evolving software (poster session). In *Proceedings of the 22nd international conference on Software engineering*, page 777, 2000.

[20] J. F. Ramil and M. M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proceedings of the International Conference on Software Maintenance*, pages 163–172, 2000.

[21] G. Robles, J. M. Gonzalez-Barahona, J. Centeno-Gonzalez, V. Matellan-Olivera, and L. Rodero-Merino. Studying the evolution of libre software projects using publicly available data. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 111–115, Portland, Oregon, 2003.

[22] G. Robles, J. M. Gonzalez-Barahona, and M. Michlmayr. Evolution of volunteer participation in libre software projects: evidence from Debian. In *Proceedings of the 1st International Conference on Open Source Systems*, Genova, Italy, July 2005. To appear.

[23] I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou. Open source software development should strive for even greater code maintainability. *Communications of the ACM*, 47(10), October 244.

[24] W. Scacchi. Free and open source development practices in the game community. *IEEE Software*, 21(1):59–66, 2004.

[25] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt. Maintainability of the linux kernel. *IEE Proceedings–Software*, 149:18–23, 2002.

[26] S. D. Serge Demeyer and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, July 2002.

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

A. About the projects analyzed

Emacs is an editor written in LISP from the 1970s but still very popular, although it was completely rewritten by Richard Stallman in the mid 80s. We assume it can be treated as a libre software legacy system, as it has been in a mature state for around 20 years now.

GCC is the GNU Compiler Collection which contains compilers for C, C++, Java, ADA, etc. Many entities are interested in the development of GCC and it has had a complicated history with several forks and merges of compilers, so large amounts of third-party code have been imported to the repository. There have been previous studies on GCC [18].

WINE is the recursive acronym for WINE Is Not an Emulator, although we could say it is a Windows emulator for UNIX systems. Its most interesting characteristic is that its development is not community-driven, so a limited amount of developers from the company who builds WINE have directly touched the repository. Wine is the unique application considered which has not achieve a 1.0 version.

The GIMP comes from GNU Image Manipulation Program and is the most-known libre competitor of Photoshop. It was started in the mid-90s by two students at Stanford who left the project after releasing the 1.0 version, but has been taken over by other developers and is today a community-driven project. The initial import into the current CVS happened December 1997 with around 39,000 lines remaining.

GTK+ is the Graphical ToolKit developed for The GIMP. It was taken over by the GNOME community and serves since then as the graphical toolkit of the whole GNOME desktop environment and not only of The GIMP. This makes it specially interesting, as there is a high number of projects which depend on it (in opposition to The GIMP). The old code base from which less than 18,500 lines remain today was imported into the current repository in December 1997.

Apache 1.3 is the most-used HTTP server worldwide. Although its development is frozen as the active branch is the Apache 2.0 one, Apache 1.3 is still the most used Apache version nowadays[8]. Apache has been the target of several research studies [14, 18].

kdelibs contains the core libraries for the KDE desktop environment. It has been selected as it is a project where a large quantity of developers have contributed to. Although it is not a graphical tool kit as GTK+ (KDE uses the Qt toolkit) it can also be seen as a project on which many others rely on.

Evolution is a libre group-ware solution by Novell which contains e-mail application, calendar, etc. It is a case sample of a corporate-driven development that has achieved to group a community around it. It has been studied before in detail by German [4, 5].

Mozilla is a well-known Internet suite which groups web navigator, e-mail client, etc. It is the libre software successor of the Netscape Navigator suite and is the canonical example for an application that has been released to the community once an ample code base already existed. This happened in April 1998; almost 154,000 lines of code remain from then. It is also one of the most studied libre software projects [1, 2, 14].

[8] Any of the Apache 1.3 versions is used by at least 50% of all httpd servers worldwide and over 66% of all Apache servers installed. Source: Web Server Survey - Server Breakdown (published March 1st 2005): http://www.securityspace.com/s_survey/data/200502/servers.html