Using Software Archaeology To Measure Knowledge Loss in Software Projects Due To Developer Turnover [*]

Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega and Jesus M. Gonzalez-Barahona
GSyC/LibreSoft
Universidad Rey Juan Carlos (Madrid, Spain)
{dizquierdo, grex, jfelipe, jgb}@gsyc.escet.urjc.es

Abstract

Developer turnover can result in a major problem when developing software. When senior developers abandon a software project, they leave a knowledge gap that has to be managed. In addition, new (junior) developers require some time in order to achieve the desired level of productivity. In this paper, we present a methodology to measure the effect of knowledge loss due to developer turnover in software projects. For a given software project, we measure the quantity of code that has been authored by developers that do not belong to the current development team, which we define as orphaned *code. Besides, we study how* orphaned *code is managed by the project. Our methodology is based on the concept of software archaeology, a derivation of software evolution. As case studies we have selected four FLOSS (free, libre, open source software) projects, from purely driven by volunteers to company-supported. The application of our methodology to these case studies will give insight into the turnover that these projects suffer and how they have managed it and shows that this methodology is worth being augmented in future research.*

Keywords: developer turnover, orphaning, software risk management, software archaeology

1. Introduction

Software development is an activity intense in human resources. The work of many developers is required to create almost any non-trivial piece of code. The lifespan of a software system can range from several years to decades. In such scenarios, the development team in charge of the software may suffer from turnover: old developers leave while new developers join the project. With the abandonment of old, senior developers, projects lose human resources experienced both with the details of the software system and with the organizational and cultural circumstances of the project. New developers will need some time to become familiar with both issues. In this regard, the time for volunteers to become core contributors to FLOSS [1] projects has been measured to be 30 months in mean, since their first contribution [11].

Although maintaining the current development team could be thought as a plausible solution to mitigate this problem, turnover is usually unavoidable. Being a highly intellectual work, developers have a tendency to lose the original motivation on the software system as time passes by, and they have the natural desire to search for new objectives. In this sense, high turnovers have been observed in most large FLOSS projects, where several *generations* of successive development teams have been identified [21]. Although these environments are partially, if not mostly, driven by volunteers, turnover in industrial environments is also high.

In this paper, we present a methodology to measure the effect of developer turnover in software projects. It is based on quantifying the knowledge that developers contribute to a project based on the number of lines of code written for it. In case developers leave, their lines become *orphaned*. The amount of *orphaned* lines can be considered as a measure of the knowledge that the

[*]This work has been funded in part by the European Commission, under the FLOSSMETRICS (FP6-IST-5-033547), QUALOSS (FP6-IST-5-033547) and QUALIPSO (FP6-IST-034763) projects, and by the Spanish CICyT, project SobreSalto (TIN2007-66172).

[1]Through this paper we will use the term FLOSS to refer to "free, libre, open source software", including 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).

project has lost with the abandonment of developers. Hence, *orphaned* lines constitute a legacy of past developers. As the amount of *orphaned* lines increases, the project may become more obscure to the current development team, since major parts of the code were written by developers who are no longer available.

Our methodology can be useful to determine the knowledge that the current development team has about the software system. This is specially interesting in the case of software maintenance, a non-trivial task that accumulates over 80% of total activity in software projects [3]. In our vision, projects with a high share of *orphaned* lines have an inferior starting position, since they need to maintain code authored by others.

We have applied our methodology to several FLOSS projects. Among them there are some led by companies, following development models which can be considered similar to those of non-FLOSS software development. Therefore results can (in part) be extended to traditional software development. However, the methodology is especially interesting for FLOSS projects, since the descriptive information obtained can be of great interest for decision-making developers. It should be noted that regarding in-house development, companies may have much more knowledge over projects they manage directly. Nonetheless, in the FLOSS world, where development is performed following distributed, highly dynamic patterns, and involving volunteers, gaining insight into a project is a major issue.

The structure of this paper is as follows. Next section presents related research. The third section introduces our methodology, based on extracting authorship information by mining the source control management system of software projects. We also highlight the new aspects that software archaeology introduces in research on software maintenance and evolution. The fourth section gives some insight into the four FLOSS projects that have been selected as case studies, while the fifth one shows the results of applying our methodology on them. Then, the methodology is discussed in detail, as well as current limitations and challenges that further research may target are also presented. Finally, conclusions are drawn in the last section.

2. Related research

Risk management during the software development and maintenance process has been a matter of research for a long time [1]. Some of the risks affecting software development are related to human resources. In DeMarco and Lister's classic *Peopleware* [4], several chapters are devoted to the management of such issues. Developer turnover is one of the most important factors to be considered. Nonetheless, in the editors notice of a special issue on software risk management [2], Boehm and DeMarco state that "Indian software managers revealed that they perceived personnel turnover as their biggest source of risk". A questionnaire addressed by Otte [18] shows that around a 12% of the total SLOCs in average is *abandoned* code. He demonstrated that projects with high levels of *abandoned* code tend to report more bugs.

Beyond pure industrial settings, recent research has focused on this issue in the case of FLOSS projects. Mockus *et al.* identified that in FLOSS projects a small, but very active number of developers perform a large part of the software development; they labelled this group as the *core group* [16]. Robles *et al.* studied the composition of the core group for several FLOSS projects over time and found that only a minority of projects shown a stable core group for many years [21]. Most projects showed a pattern where *generations* of developers successively take the lead for a limited amount of time, generally in the range of two to four years. Michlmayr *et al.* introduced the measure of half-life for the human resources of a project [15]. They analyzed the population of developers participating in a project and identified the point in time when only half of them are still active. In the case of Debian, the half-life observed was 7.5 years. However, other studies have pointed out that packages maintained by Debian developers who abandoned the project were later maintained by others, showing a natural "regeneration" process [22].

In general, developers who leave a project somehow carry with them their knowledge about it. If that knowledge is not shared with other developers, it can be considered as disappearing from the project. How to characterize that knowledge is a common case of study. Hutchison [13] concludes that "The knowledge system is not the individual but the entire history of problem solving teams in which individuals actively participate". Ge *et al.*, focusing on FLOSS projects, state that the expertise is not centered in just one member [7]. They conclude that no single member is the owner of the whole knowledge, even for specific parts of the code. However it is a challenge to recover all this expertise (knowledge) when developers leave, because it is usually not recorded anywhere. In the case of FLOSS projects, it is recorded only in part, in mailing lists archives, source code management or bug tracking systems, where source code, new ideas or comments are registered. All this common knowledge can help the "regeneration" of knowledge once a given developers leaves.

German observed empirically that developers have a tendency to work on specific parts of their software project, so that it is easy to determine the files in which they work. This behaviour has been labelled as "developer territoriality": analogous to some species, developers have their own "territory" [9].

FLOSS projects are usually peculiar in that they depend on the work of many volunteers (even when some times hired developers also join the effort). Much of the *bazaar* style of development roots on attracting third party contributions, and in integrating new developers in the project [20]. This introduces new challenges in the management of human resources in a software project [6, 24], as well as in the areas of quality and reliance [14].

With respect to our methodology, which is based on information retrieved from mining the source code repositories of software projects, some previous research lines exist. For example, Rysselberghe *et al.* [23] proposed a method to study the changes performed on a software system hosted in a source code repository. The authors display a horizontal line for every file, and changes to files are expressed by dots in that line. This visually identifies which parts of the system are modified over time, and also which files change simultaneously.

Gîrba *et al.* aggregate some information to identify which developers implemented which parts of the system [10]. They introduce colors for authors, assigning a color to horizontal lines depending on who contributed most to a file. This visualization can be used to identify development patterns.

3. Theoretical framework

The base concept behind the methodology proposed in this paper is *orphaning*. Briefly, orphaning can be defined as the percentage of *orphaned lines* with respect to the total number of source lines of code (SLOC in the following).

The retrieval of information related to the origin of each line is the basis of the study. "Who is the owner?" and "when was it modified?" are the two main question to be answered.

If we consider any point in the past, some lines are authored by active committers (those who still contribute to the project), while some others are authored by non-active committers (those who already left the project). The latter set of lines is defined as *orphaned*. And by definition, those lines tend to disappear with time, as the software experiments further changes.

The life of orphaned lines is different depending on what is happening in the project. For instance, if developers are focused on adding new functionality, it is likely that orphaned lines are only seldom touched. On the contrary, in a project performing an intense code refactoring process, many orphaned lines will disappear. However, in both cases the proportion of orphaned lines to total number of lines can evolve in a similar way. Therefore, a high (or low) orphaning does not necessarily mean anything about the past history of the project. On the contrary, it can reflect current trends in it.

The idea of considering orphaned lines brings this study into the field of *software archaeology*[2], which can be seen as an extension to *traditional* software evolution. As Hunt states it [12], researchers working in this field deal, as archaeologists, with artifacts remaining from the past.

Although the concept of software archaeology was conceived with large legacy systems in mind, it is valid for any type of software with independence of age and size. Its main basis is the need that developers have, when maintaining a software product, to deal with code that was modified in possibly many points over time by many different developers.

The idea of relating archaeology to software maintenance can be tracked back to the OOPSLA 2001 Workshop on Software Archeology, organized by Ward Cunningham et al. The promoters of this workshop had a set of assumptions, being the first one the rationale for using the archaeology concept:

> "[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."

Up to now, growth studies in software evolution have been based on considering snapshots of a software system over time. As depicted in figure 1, for the software evolution metaphor the software engineer is aware of how the software has changed. Based upon these observations, some conclusions can be derived.

In general, the view provided by software evolution implies a situation which is not that important for the

[2]In American English *archeology*, comes from the Greek meaning arqaios (ancient) and lógos (word/speech).

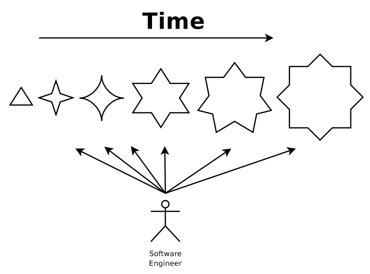

Figure 1. Software evolution point of view.

maintenance of a software system. It is the current state of the sources which becomes the most important issue, being all previous states less important, especially if nothing is left behind. Figure 2 shows the approximation of software archaeology; the software engineer looks backwards *through* the current state of the software. The current state of the software will be heavily influenced by previous states of the software system, but only those that have persisted are of interest. That is, code and other artifacts that cannot be found in the latest version are uninteresting.

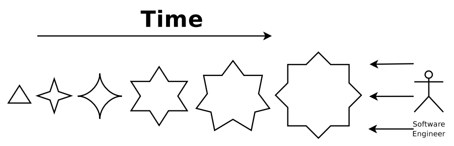

Figure 2. Software archaeology point of view. The software engineer views from the current state of the software into the past.

The human team in charge of the maintenance process should also be considered to characterize the aging of a system. A maintainer who has some knowledge about a system will maintain it better than somebody with no experience with it. Introducing these aspects into our archaeology framework is not difficult, as changes are associated to the humans performing them. Hence, we can expect that developers who introduced code which still persists in the system have a certain advantage when maintaining it. The opposite is also true: losing a developer with a large experience contributing to the project, (i.e. who has done many changes), will affect the maintainability and aging of the software.

4. Methodology

The methodology considers projects that store source code in a SCM (source code management) system, such as CVS or SVN (Subversion). Therefore, the whole lifespan of projects can be analyzed.

4.1 Definitions

We define the following concepts:

- *Line of code*: line finishing with a newline or end of file character, and containing at least one character that is not a blank space or comment.
- *Author*: Original developer of a line of code.
- *Commit*: Change(s) made by a committer to source code. In a change, lines can be added or deleted (modification is considered as a deletion followed by an addition, which is enough for tracking the history of authors of lines).
- *Current state of the code*: Lines of code included in the current version of the software. This is the set of lines added minus those deleted.
- *Past state of the code*: Lines of code included in past versions of the software. This is the set of lines added minus those deleted before that date. In other words, it is the source code base obtained in a given date. The current state of the code is, therefore, the past state of the code when the specified date is today.
- *Committer*: Developer with write access to the SCM system, who can therefore modify the files stored in it by performing commits. A committer can be the author of the source code he commits or not, as projects do not provide write access to anyone. Hence, third-party contributions are not committed directly by their original authors, but by a committer.
- *Non-active committer*: Committer with no activity in the SCM system since a given date in the past. The timespan for which a committer has been considered as non-active in this paper is one year, although other values could be considered.
- *Orphaned Line*: Line of code that can be found in the current state of the code, but which was committed by a non-active committer.

Thanks to the metadata stored in the SCM, it is possible to determine when a line was introduced, who introduced it, and its content (needed to identify that it is not a blank line or a comment).

In the rest of the paper we assume that committers have *complete knowledge* over the lines they commit, *i.e.*, that a committer is fully aware of all the lines that he introduces into the SCM system. This means that, although not necessarily being the authors, committers have a similar capacity of understanding, modifying and explaining the introduced lines as if they had authored them.

4.2 Process

The process consists of several steps that have been implemented in a software program called *carnarvon*, which works with CVS or SVN repositories. *Carnarvon* downloads (i.e., checks out) the current state of the code, and then it retrieves information about the committer who introduced each of the source code lines found in source code files.

To obtain the information about commmitters, the *annotate* command of the SCM is used[3]. Table 1 shows a (slightly modified) excerpt of the annotated output for the src/keymap.c file from GNU Emacs, extracted from its CVS repository. The structure of the CVS annotate output is as follows. For each line, the first column contains the file revision in which it was introduced as it is now. After it, within brackets, the username and date of the commit. Finally, the content of the line. The structure of the SVN or other SCM output of the *annotate* (or equivalent) command is slightly different, but also contains the information relevant to this study.

The analysis considers only files with source code. Therefore, documentation or translation files are ignored in this process. *Carnarvon* stores all this information in a database, in which each relevant line corresponds to an entry in a table. The most important fields of that entry are:

- **committer_id** - Committer who introduced the line.
- **file_id** - File containing the line.
- **date** - Date when the line was introduced.

If the current version of the software is considered, information is obtained about how old are each of the lines that compose the software, as well as who committed them. This allows, for instance, to know how many of the current lines of code are older than 1 year, or how many of those, that are currently part of the software were committed by developers that left the project 1 year ago (or earlier), are *orphaned* lines.

Since this can be done for any point in time, the same study can be performed for past states of the code. For instance, the version of the code corresponding to one year ago can be checked out, quantifying how many lines were, at that moment, at least one year old (i.e., were committed more than two years ago and were still present one year ago), or how many of them were *orphaned* lines then (i.e., were part of the past state of the code and were committed by developers leaving the project at least 2 year ago).

To perform an evolutive analysis, several tables (one for each iteration) are constructed. *carnarvon* also stores information about committers, and a hierarchical structure of the directory tree and files.

5. Case studies

We have applied our methodology on four FLOSS projects, which will be presented in this section. All of them are part of the GNOME platform, a desktop environment for UNIX-like systems. Hence, they share a common development culture (rules, procedures and ways to join a project, among others) [8]. But they differ on some key issues that made them worth to be studied, specially with respect to the composition of their development teams:

- Evolution is a groupware solution that combines e-mail, calendar, address book and task list managements functions. It is a good representative of company-driven FLOSS projects, although with a large quantity of contributions by the community. It was the flagship application of Ximian, before it was taken over by Novell. Since then, its maintenance has been outsourced to a Novell team in India.
- GIMP (GNU Image Manipulation Program) is an outstanding graphics editor. It is more than 12 years old, although it uses a SCM system only since 1998. The original authors of the GIMP left the project, and a new generation of volunteers has taken it over.
- Evince is a document viewer for PDF and PostScript. It started as a rewrite of *GPdf* —in fact, the commits before 2004 are from the original source code base. So, it is a good representative of a product that is completely taken over by a new team in order to be revamped.

[3]The *annotate* has several alias, being *blame* and *praise* other possibilities.

```
[...]
1.246    (pj      13-Nov-01): /* Optional arg STRING supplies menu name for the keymap
1.246    (pj      13-Nov-01): in case you use it as a menu with `x-popup-menu'. */)
1.246    (pj      13-Nov-01):      (string)
1.8      (rms     11-Sep-92):      Lisp_Object string;
1.8      (rms     11-Sep-92): {
1.8      (rms     11-Sep-92):   Lisp_Object tail;
1.8      (rms     11-Sep-92):   if (!NILP (string))
1.8      (rms     11-Sep-92):     tail = Fcons (string, Qnil);
1.8      (rms     11-Sep-92):   else
1.8      (rms     11-Sep-92):     tail = Qnil;
1.1      (jimb    06-May-91):   return Fcons (Qkeymap,
1.137    (rms     13-May-97):     Fcons (Fmake_char_table (Qkeymap, Qnil), tail));
1.1      (jimb    06-May-91): }
[...]
```

Table 1. CVS annotate output for the src/keymap.c file of the GNU Emacs project (excerpt).

- Finally, Nautilus is the official file manager for the GNOME desktop. It is an important project in GNOME (it manages the desktop, for instance) and has also a long history: it started as the flagship application of a company that went bankrupt in 2001 (Eazel) and has been taken over since then by the GNOME community.

6. Results

To present the results of applying the aforementioned methodology on the projects selected as case studies, we start depicting the evolution over time of the number of *orphaned* lines of code versus the total lines of code in the projects. Then, the relative (*orphaned* vs. total lines) evolution will be shown. Finally, the evolution of *orphaned* code is presented, to observe whether the code from non-active developers is removed quickly, progressively, or not at all.

6.1 Total lines vs. *orphaned* Lines

Figures 3 and 4 show the evolution of the number of lines of source code, also known as the software growth curve, and the evolution of the number of *orphaned* lines for the four projects under study. To obtain these graphs we sampled the repository every month, obtaining a monthly snapshot that we consider in each case as the past state of the code.

The evolution for GIMP and Evolution can be seen in Figure 3. The growth curve of GIMP has been oscillating around 800,000 lines of code for the last 10 years, with an tendency to increase from 1998 to 2001, then some net code removal until 2003, and from then on a timid growth until today. The evolution of the *orphaning* curve can be split into two parts. Before mid-1999, there was almost no *orphaning*. Then, one of the major contributors to GIMP left the project and almost half of the source code became *orphaned*. The amount of orphaned code has been decreasing since then. Since the amount of total code for the project has almost remained the same, we can state that the amount of *orphaned* code also decreases in relative terms.

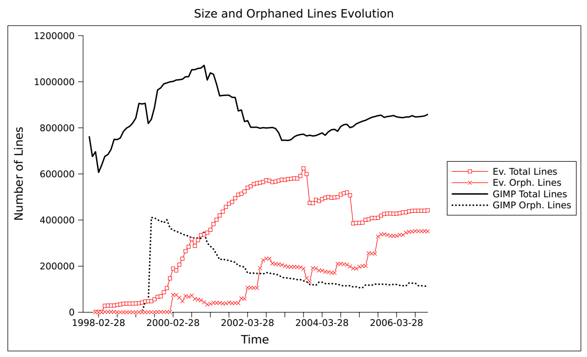

Figure 3. Evolution of total size and number of *orphaned* lines for Evolution and GIMP.

In the case of Evolution, the growth curve has an S-like shape until 2004. After that point, a significant quantity of code has been removed twice from the main trunk, causing a fall in the total number of lines. The evolution of *orphaning* also has several phases. Up to 2000, almost no *orphaned* lines exist. Then, until 2002, a small number of them appear that remain almost constant in time. There is a first raise in the number of *orphaned* lines in 2002 which gets stabilized until a second big jump in 2005. All in all, in relative terms, the *orphaned* curve tends to become closer to the software growth curve as time passes by.

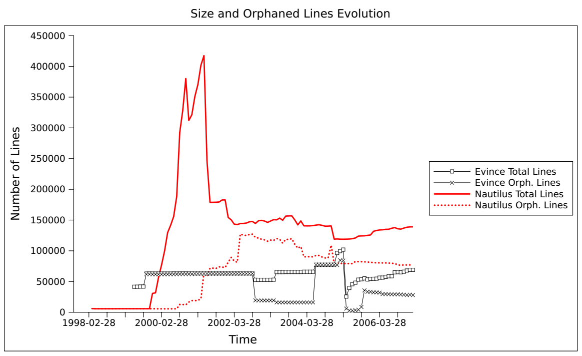

Figure 4. Evolution of total size and number of *orphaned* lines for Evince and Nautilus.

Figure 4 provides the same information, but for Evince and Nautilus. Regarding size, Evince is a medium-sized project of around 100,000 lines of code. Early in 2005, the old Xpdf part was removed, resulting in a loss of over half of its size and a mild posterior growth that hints to the addition of new functionality included in Evince in recent releases. The *orphaning* curve gives plenty of information that allows us to better understand the history of Evince. Between 2002 and 2004, a group of developers worked on Evince, but then the project was taken over by a different group that decided to remove a large part of the code base, to maintain part of the old code base and to develop new functionality.

With respect to Nautilus, we find a similar behaviour to that of Evolution. In this case large additions of code can be observed, as well as large removals, shown by vertical lines in the software growth curve. Since more than 5 years ago, Nautilus has an almost stable amount of lines of code. Regarding its *orphaning* curve, a steady raise can be observed, up to the point that in late 2002 all lines in the project were *orphaned*. Since then, the number of orphaned lines tended to decrease.

Figure 5 shows the evolution over time of the *orphaned* lines, compared to the total size of the project. Many lessons can be learned about the projects studied here. Regarding GIMP, the most important event happens when a main developer abandons the project back in 1999. Since then, it has managed to lower the share of *orphaned* lines gradually. So, even when that abandonment was a great loss in knowledge, the project has achieved to counter it and to have values of *orphaning* that are currently close to 20%. This can be seen as a low risk value, since a vast majority of the code is known to the developers.

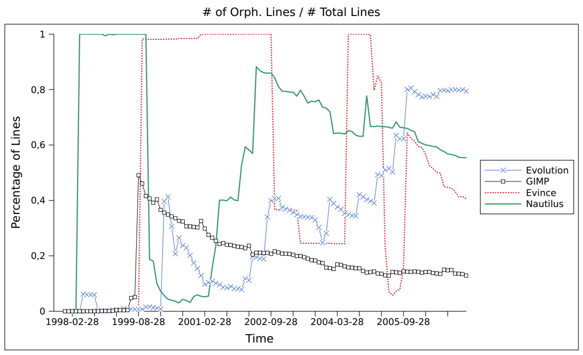

Figure 5. Evolution of *orphaned* lines relative to the total number of lines.

The opposite behaviour can be observed in Evolution. The project featured low values of *orphaning* during 2001 and 2002, coincident with a high activity by Ximian. Since then, the share of *orphaned* lines has been growing steadily, up to a point where today almost 80% of the code has been introduced by currently non-active developers. This supposes a high risk in the maintenance of the project, as the team now in charge is not the one that introduced the code into the repository.

Nautilus has a mixed behaviour. Up to mid-2002 it shows a high risk due to large amounts of *orphaned* code when it achieved values of over 80%, but the trend since then has been to lower the share of *orphaned* code, with current values lying under 60%. As the software size has not grown much during the last years, this may be an indicative for maintenance work such as refactorings. In any case, compared to Evolution, it seems that the current team in charge of the maintenance of the project has better knowledge about it, which would mean that the risk factor is lower.

Finally, Evince shows the shape of a project that has been completely abandoned (we find orphaning shares of 100%) and taken over several times. Since its last abandonment (in early 2005), a large effort has been devoted to lower the amount of *orphaned* lines; so, values of *orphaning* as low as 10% could be measured in mid-2005. But, again, as an important contributor left the project, much parts of the code became *orphaned* in late-2005 with values over 60%. In any case, the current trend is to lower the amount of *orphaned* code, in this case by adding new functionality (new code) and maintaining the old one. Regarding risk, we have to consider that Evince is the smallest of all the projects under study, which makes the amount of *orphaned* lines of code not such a a major concern, while it explains much of its instability as well.

6.2 Evolution of *orphaned* lines

So far, the amount of code that can be assigned to non-active committers has been considered. In this subsection, the attention is focused on what happens to that code: how does *orphaned* code evolve over time. By definition, the number of *orphaned* lines is a decreasing function as non-active developers cannot add new lines. But, as in the case of the half-life measure for radioactive elements which was applied to the abandonment of a population of developers in FLOSS projects [15], these curves could give us further insight about what happens to code from non-active developers.

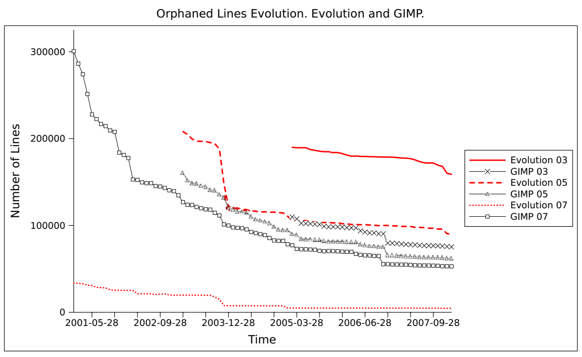

Figure 6. Evolution of *orphaned* lines for the Evolution and GIMP projects. The lines have been selected for three periods: those older than 7 years (07), those older than 5 years (05) and those older than 3 years (03).

Figure 6 depicts the pace at which these orphaned lines are removed in Evolution and GIMP. Lines marked as *orphaned* three, five and seven years ago, have been considered. For each of these groups, their decrease over time has been studied. GIMP and Evolution show a different story in this respect. Evolution shows a gap due to code removal in late 2003, and only small decreases over time. In GIMP *orphaned* lines are removed at a higher pace. This is indicative of a higher effort in maintaining *orphaned* code in GIMP than in Evolution, which is consistent with the results discussed in the previous subsection.

Figure 7 depicts the same curves for Nautilus and Evince. In the case of Evince lines from non-active committers are removed with time, suggesting the idea that "if you are not participating in the project your code is

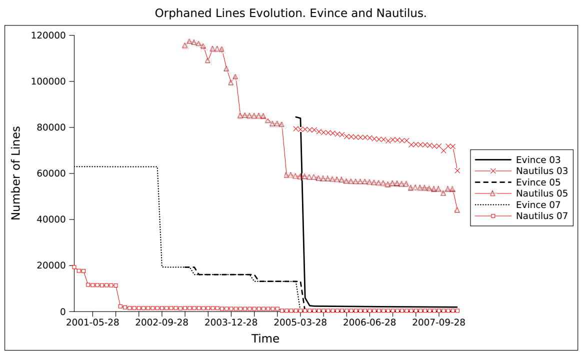

Figure 7. Evolution of *orphaned* lines for the Nautilus and Evince projects. The lines have been selected for three periods: those older than 7 years (07), those older than 5 years (05) and those older than 3 years (03).

removed". Nautilus, on the other hand, is similar to Evolution, since decreases in *orphaned* lines are mainly due to removal of large parts of the system. Besides that, the decline in the number of *orphaned* lines is minimal.

7. Discussion

In this section limitations and future challenges will be discussed.

7.1 Limitations

Several limitations have been detected during the application of the methodology:

- Presence of a *gate-keeper*: performing changes (commits) to a repository is only possible for those with write access. Depending on the policy, projects may have a small set of developers who are in charge of committing all the changes. This will skew the results produced by our methodology in two ways: first, the committer will be assigned code that he may not know, and second, even if the committer is non-active the original author may be present avoiding that the lines of code become *orphaned* (impersonation). This limitation has a technical and organizational solution: some source code managements (SCM), like *git* but not CVS or SVN, allow to make a distinction between the author and the committer. If projects introduce a policy to indicate this, then this would counter the effect of this limitation. On the other hand, gate-keepers are usually in charge of reviewing the code they commit, so the assumption that they know it is not a weak one; in addition, many casual contributors do not participate for long in FLOSS projects, thus the problem of impersonation will mostly be insignificant.

- Absence of *full memory*. Our methodology assumes that authors have a complete awareness of their own code. This may be true for code written in recent times, but is a risky assumption for older code, especially in large software systems. A way of addressing this issue would be to treat lines of code differently depending on their age.

- Introducing *software comprehension*. By now, we consider all lines of code as equal, not attending to quality factors such as how easy it is to understand or how well it is structured. A good example of this is that, poorly-structured *spaghetti* code from an active committer could be by far less easy to maintain than well-written *orphaned* code. Solving this problem is not an easy task; methods proposed in the area of software comprehension or including complexity information could be added to target it.

7.2 Future challenges

In addition to the limitations, we have also found some challenges for future research that would enhance the methodology:

- The information provided by SCM systems is on a line by line basis. However, an optimal granularity for the methodology would be at the function (or method) level. The rational for this is that developers do not think on a line by line basis, but consider larger pieces of code with some semantic meaning. The use of functions (methods) as the minimum granularity level will have to deal with ways of summarizing the information of several lines into a unique parameter, that should be defined with care. The authors of this paper are already working on this idea.

- A major drawback of the methodology is that the assumption that *orphaned* code is more difficult to maintain than non-*orphaned*, although plausible, has not been demonstrated so far. The use of information from other data sources, such as bug notifications, that could be matched to the changes to the repository, would allow to compare *orphaned* code and non-*orphaned* for such evidence. Another interesting topic of research would be to analyze if *old* code (i.e., code that was introduce long time ago) is more maintainable than *young* code (i.e., code submitted recently). The authors of this paper have found some relevant literature about software aging [19] and code decay [5], but none of these studies present conclusive results about this question. Also, Otte [18] has demonstrated that projects with high levels of *abandoned* code tend to report more bugs related to quality issues. However, there are some differences between the definition of *orphaned* and *abandoned* code.

- It would be useful to match the methodology with the traditional metrics for assessing levels of maintenance [17], or with some existing models to predict maintenance effort on FLOSS projects [25].

- More software projects, from different environments should be studied to enrich the methodology and provide new insight into the facts that can be extracted from it.

8. Conclusions

The turnover of developers is sometimes a big problem for companies and FLOSS projects. Generally speaking, the time to learn how the project works causes great productivity losses, which are unavoidable when senior developers leave the project and are substituted by others, new to the project, or to the parts of the code they have to maintain.

The methodology presented in this paper tries to measure quantitatively the effect of turnover in software projects by using the amount of code contributed by a developer. Using the described methodology we have studied four FLOSS projects and extracted some conclusions from our preliminary analysis. In this regard, we have observed several situations, ranging from a project that that tries hard to keep the code known to the current development team (GIMP), to a project in which a high turnover has caused that the current maintenance team has authored a very small portion of the total code (Evolution). In between, Evince provides an example of a project where developers retake other's code and rewrite it, while Nautilus illustrates the case of an abandoned project which is taken over by a new team.

The analyzed cases allows to state that the use of *orphaned* lines, especially in relative terms to the total number of lines, is a good measure of the *health* of the software project. This information could be very valuable for managers, especially for those who do not have insight into the specifics of the development process, a common situation for many managers that use, but do not participate in FLOSS projects. Insiders could use this information to avoid risks (e.g. modules plenty of *orphaned* lines where the current *core* team has never worked on) or to strengthen the maintenance activities of a project (e.g. detecting those ”dead” modules).

All in all, the presented methodology, based on the idea of *software archaeology*, provides useful information to address the issue of developer turnover in software projects, even if some limitations, both technical (*gate-keeper* effect, *granularity*, etc.) and conceptual (*old* code *vs. young* code, the effect of the quality, *full memory*), have still to be dealt with in the future.

References

[1] B. W. Boehm. Software risk management: Principles and practices. *IEEE Softw.*, 8(1):32–41, 1991.

[2] B. W. Boehm and T. DeMarco. Guest editors' introduction: Software risk management. *IEEE Software*, 14(3):17–19, 1997.

[3] S. A. Conger. *The New Software Engineering*. International Thomson Publishing, 1994.

[4] T. De Marco and T. Lister. *Peopleware : Productive Projects and Teams, 2nd Ed.* Dorset House Publishing Company, Incorporated, 1999.

[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*, 27(1):1–12, 2001.

[6] K. Fogel. *Producing Open Source Software : How to Run a Successful Free Software Project*. O'Reilly Media, Inc., 2005.

[7] X. Ge, Y. Dong, and K. Huang. Shared knowledge construction process in an open-source software development community: an investigation of the gallery community. In *ICLS '06: Proceedings of the 7th international conference on Learning sciences*, pages 189–195. International Society of the Learning Sciences, 2006.

[8] D. M. German. The GNOME project: a case study of open source, global software development. *Journal of Software Process: Improvement and Practice*, 8(4):201–215, 2004.

[9] 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.

[10] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of the International Workshop on Principles in Software Evolution*, pages 113–122, Lisboa, Portugal, September 2005.

[11] I. Herraiz, G. Robles, J. J. Amor, T. Romera, and J. M. González-Barahona. The processes of joining in global distributed software projects. In *GSD '06: Proceedings of the 2006 international workshop on Global software development for the practitioner*, pages 27–33, New York, NY, USA, 2006. ACM Press.

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

[13] C. Hutchison. Personal knowledge, team knowledge, real knowledge. *EUROCON'2001, Trends in Communications, International Conference on.*, 1:247–250 vol.1, 2001.

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

[15] M. Michlmayr, G. Robles, and J. M. Gonzalez-Barahona. Volunteers in large libre software projects: A quantitative analysis over time. In S. K. Sowe, I. G. Stamelos, and I. Samoladas, editors, *Emerging Free and Open Source Software Practices*, pages 1–24. Idea Group Publishing, Hershey, Pennsylvania, USA, 2007.

[16] 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.

[17] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344, Nov 1992.

[18] T. Otte, R. Moreton, and H. D. Knoell. Applied quality assurance methods under the open source development model. In *COMPSAC*, pages 1247–1252, 2008.

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

[20] E. S. Raymond. The cathedral and the bazar. *First Monday*, 3(3), March 1998. `http://www.firstmonday.dk/issues/issue3\_3/raymond/`.

[21] G. Robles. Contributor turnover in libre software projects. In *Proceedings of the Second International Conference on Open Source Systems*, 2006.

[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*, pages 100–107, Genoa, Italy, July 2005.

[23] F. V. Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *International Conference on Software Maintenance*, pages 328–337, 2004.

[24] J. Sandred. *Managing Open Source Projects*. Wiley Computer Publishing, 2001.

[25] L. Yu. Indirectly predicting the maintenance effort of open-source software: Research articles. *J. Softw. Maint. Evol.*, 18(5):311–332, 2006.