Modification and Developer Metrics at the Function Level: Metrics for the Study of the Evolution of a Software Project

Gregorio Robles
Universidad Rey Juan Carlos
grex@gsyc.urjc.es

Israel Herraiz
Technical University of Madrid
israel.herraiz@upm.es

Daniel M. Germán
University of Victoria
dmg@uvic.ca

Daniel Izquierdo-Cortázar
Universidad Rey Juan Carlos
dizquierdo@gsyc.urjc.es

***Abstract*—Software evolution, and particularly its growth, has been mainly studied at the file (also sometimes referred as module) level. In this paper we propose to move from the physical towards a level that includes semantic information by using functions or methods for measuring the evolution of a software system. We point out that use of functions-based metrics has many advantages over the use of files or lines of code. We demonstrate our approach with an empirical study of two Free/Open Source projects: a community-driven project, Apache, and a company-led project, Novell Evolution. We discovered that most functions never change; when they do their number of modifications is correlated with their size, and that very few authors who modify each; finally we show that the departure of a developer from a software project slows the evolution of the functions that she authored.**

***Keywords*-functions; metrics; software evolution; software maintenance; mining software repositories**

I. Introduction

Historically, software evolution has been studied at the file level. Lehman already used in his *laws* of Software Evolution the number of files (in his nomenclature *modules*) as a measure of system growth [1]. Other authors have used SLOC (source lines of code) for the same goals [2].

Our position in this paper is that by moving from the file/SLOC (physical) to the function (physical and semantic) level, we can gain better understanding of the evolution of a software project. In our point of view, considering functions is closer to the way developers work and conceive a software system. And, as we will show in this paper, new insights can be gained by doing so.

When starting with this research, we had the intuition that the relationship that developers have with functions is stronger to the one they have with files. We have therefore specifically addressed two metrics from software evolution research that have already been studied at the file level, *modification patterns* and *developer territoriality* (also known as *code ownership*), but this time considering a granularity of functions. While the former is concerned with the way that functions are changing (frequency of changes, number of changes and temporal frequency of changes), the latter is related to the developers that change the functions, in particular if a "special" relationship exists between its creator and the function, and if this relationship affects the evolution of the function.

The remainder of this paper is structured as follows: next, five hypotheses will be presented and briefly explained. We describe the methodology used to address our hypotheses in the fourth section and apply it to the case studies briefly presented in the fifth. The next section contains the results and their discussion. Finally, conclusions are drawn. Due to the limited space for full papers at this workshop, a specific section with related research has been omitted, although there is a vast literature in the areas of software evolution, modification patterns and code ownership.

II. Hypotheses

To demonstrate the advantages of the use of functions as basis for software evolution metrics, we first have to understand how functions evolve, and second how the evolution of functions is affected by their authors. We ground our study in the following hypotheses:

- Hypothesis #1: Most functions rarely change. We believe that most functions are created, they are modified few times to get them "right", and then they do not evolve any further.
- Hypothesis #2: As functions age, they are less likely to be modified. The older a function is, we hypothesize, the less likely somebody will be willing to modify it. Also, functions that implement their functionality properly do not need to evolve (except if bugs are found). This hypothesis is related to #1.
- Hypothesis #3: Most lines of a function are authored by very few developers. It is acknowledged that developers tend to keep ownership of their code [3], and therefore most functions will be written by very few developers. Furthermore most lines of code in a function will be contributed by very few developers (and for most, only by its original author).
- Hypothesis #4: Orphan functions are less likely to be modified. When the original maintainer of a function is no longer a contributor (the function is "orphan"), such function tends to evolve slower than functions whose authors are still participating in the project ("non-orphan").

III. Methodology

A. Identification of Functions

To study the evolution of individual functions we needed to extract each function from each file revision and compare to its predecessor. The procedure we used was, for every revision of every file ever present in the repository (including those that were deleted):

- Using `exuberant tags` extract the location where each definition starts.
- For each function defined in a file find its ending location. The end of a function is assumed to be the location of the last closing brace before the next definition (macros are not considered a definition, as macros can appear in the middle of a function).

B. Analysis of the Evolution of Functions

We were also interested in knowing when the change was only to comments or whitespace (such as reindentation). For this reason we also computed what we call *clean* versions of the functions. This was done by preprocessing each file: first comments were removed (using `mangle`), and then it was reindented (using `indent`). Functions were then extracted as described above.

We compared then the functions present in each file revision to its predecessor, and the result was a list of all unchanged, modified, added and removed functions. These data allowed us to track when a function was added, when it had been changed (skipping changes to only their comments or whitespace), and when it had been deleted. We also tracked its size. We uniquely identified a function by its name and file where it was found.

However tracking when a function is created is not trivial. For instance, a function could be moved, copied, cloned, imported from an external source, or the result of a merge of a branch in the repository. In our study we discovered that some functions are moved from one place to another (potentially renamed) leaving the older version intact (which might be removed later); also sometimes functions are imported from an external source (such an external patch) and the version control system has no history of the function previous to such import.

For this reasons we decided to select a subset of functions that did not suffer any of these changes, and which we could track from their creation. We call this the set of *Selected* functions. This set is created using the following heuristic:

- Start with all the functions that exist in the last version of the system.
- Eliminate functions that were added in the first revision of a file. We expect this will eliminate functions that were in files that were renamed or moved; it will also eliminate those functions that were imported into the project.
- Eliminate functions with the same name that were present in another file, and then removed. We expect to catch some of the functions that were individually moved (with the same name).
- Eliminate functions that have been added at the same time as at least another function was deleted (not necessarily with the same name). We expect that this will take care of renamed and refactored functions.
- Eliminate all functions that were added in a transaction (commit) that added more than 10 functions. We decided that if the transaction adds too many functions then it is very likely that this is a merge with code that had been developed somewhere else (this includes a merge of a branch).

C. Analysis of the Ownership of Functions

Based on the meta-data that versioning systems offer, we are able to follow on a per-line basis who and when a line was last modified. Versioning systems provide an option (`blame` or `annotate`) which shows the revision where each line was last modified, giving the date and the committer responsible for the changes to each line.

Table I presents an excerpt of the output of the `annotate` option, which contains following fields: the first column is the file revision when such line was last changed, followed, in between parenthesis, by the username of the person who committed this revision and the date of the revision; and finally the content of the line from the file.

The analysis sequence starts by downloading the repository at a given time, obtaining a local copy of the repository. Then the source code is parsed and cleaned of comments, blank lines and errors, and inserted into a database server which is queried for statistical information on the data set. In addition, functions are identified using `exuberant tags`, and their starting and ending lines stored.

Combining the authorship information given at the line level and the location of the functions, we are able to compute the contribution of each author to a given function. We refer to the author with the largest proportion of lines of a function as its *primary author*.

The versioning system also provides information on the activity of developers. We are interested in the date of the last commit performed, as we assume that if there is no recent activity (i.e., a commit) by a developer, then he has *abandoned* the project. Functions whose primary author has abandoned the project will be referred as *orphan* functions. In this paper, we have chosen at least 36 months (3 years) as the time of inactivity to consider all functions by a developer as *orphan* functions. With this value, we believe to have a sufficiently long time span to assure that the *owner* of the functions is not part of the project. We leave for further research the study of a more accurate value, which we have the intuition is smaller then the one used.

```
[...]
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 I
EXAMPLE OF A VERSIONING SYSTEM ANNOTATE OUTPUT.

If we apply the aforementioned concepts to the excerpt shown in Table I, we see that the `annotate` output lists 13 lines by three different authors. Following our methodology, we will not consider the first two lines by *pj* as they are commented or blank lines. The function itself contains eleven lines, one by *pj*, eight by *rms* and two by *jimb*. Thus, *rms* would be considered the *primary author* of the function as he has contributed most to it (with 72% of the lines of the function). As the date of the last commit by developer *rms* is from March 2001, we would consider this function to be *orphan*.

IV. CASE STUDIES

We have selected two well-known free software projects as case studies: Novell Evolution and the Apache httpd web server. These projects have been under development for over 10 years now and count with a large amount of contributors and users, with a large number of committers (69 for Apache, and 395 for Evolution).

The Apache web server is the most used web server; the version under study (1.3) is basically in its maintenance phase. Apache is a project that is primarily developed by volunteers. Novell Evolution is a groupware solution for the GNOME project that provides e-mail client, calendar, contacts and task management similar in scope and functionality to Microsoft's Outlook. Although a free software project, Evolution is backed by a group of professional developers hired by Novell. At the time of this study Apache has approximately 86 kSLOC, 8,021 commits and 29,999 revisions. For Evolution the totals are: 307,883 kSLOC, 28,570 commits, 113,915 revisions. Both systems are written primarily in C.

V. RESULTS

A. Hypothesis #1: Most functions rarely change

To test this hypothesis we looked at the number of times any given function has changed[1], and at the quartiles of the distribution of that parameter. Table II shows the quartiles for the cases of Apache and Evolution, both for all the functions and the *Selected* functions. As that table shows, for the case of Apache, 75% of the functions have been changed 6 times or less. If we consider only the *Selected* functions, 75% have not been changed. For Evolution, the numbers are similar. In that project most functions are rarely changed, and very few functions are modified many times, supporting our hypothesis.

	Quartile (%)	0	25	50	75	100
Apache	All	0	0	0	5	229
	Selected	0	0	0	0	50
Evolution	All	0	0	1	3	634
	Selected	0	0	0	0	129

Table II
QUARTILES OF THE NUMBER OF CHANGES PER FUNCTION. MOST OF THE FUNCTIONS RARELY CHANGE. FOR INSTANCE, IN THE SET OF ALL THE FUNCTIONS OF APACHE, 75% OF THE FUNCTIONS CHANGE 5 TIMES OR LESS.

We were also surprised to find that in Apache 33% of *Selected* functions (138 out of 408) that were never updated. In other words, they were added, and never modified again. For Evolution the proportion was 61%! (1955 out of 3175). We doubt that all of them are dead-code. We speculate that developers make sure most of the code is well tested before they commit it to their projects.

Observation #1: Most functions change very few times (or none).

B. Hypothesis #2: As functions age, they are less likely to be modified

To test this hypothesis we considered only functions alive (non-deleted) for the set of all functions. Due to hypothesis #1 we know that the distribution of the number of changes of functions is inverse exponential, so we want to eliminate the effect of the number of changes of a function while testing this hypothesis. We therefore considered functions that have been modified at least once after they are created (and have not being deleted to this day). For these functions we computed their *days-to-last-change*: the number of days

[1]Note that we do not consider the initial addition of a new function as a change, as the versioning system does.

between the creation of a function and its last modification. The density distributions of days-to-last-change for both projects are depicted in Figure 1. As it can be seen, most functions are changed during their early days. There are however a handful that are modified while very old. This is expected: there are some areas of the software that need to be updated all the time: for example areas where new functionality is inserted into the system (such as GUI callbacks to such functions).

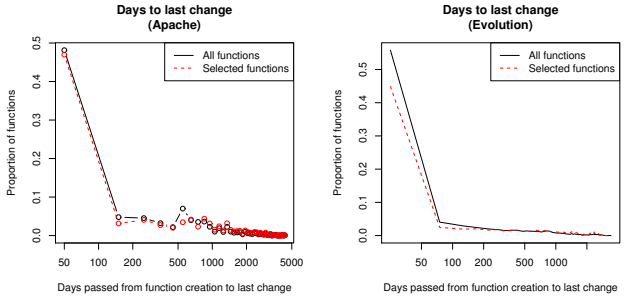

Figure 1. Density distribution of the days-to-last-change for non-deleted and *Selected* functions in Apache and Evolution. Most functions are modified early in their lifes.

We were surprised, however, that the distribution of all non-deleted functions and *Selected* functions is almost identical for both projects.

Observation #2: The older a function, the less likely it will be modified.

C. Hypothesis #3: Most lines of a function are authored by very few developers

To test this hypothesis we used two approaches. First, we plotted the distribution of the number of authors that each function has had. This is shown in Figure 2. The distributions are inverse exponential for both projects, and for both sets of functions in both projects.

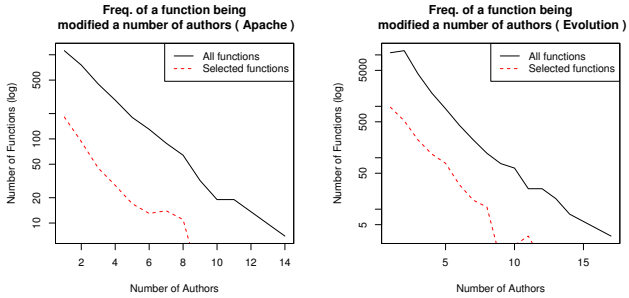

Figure 2. Distribution of number of authors for non-deleted and *Selected* functions (in logarithmic scale) in Apache and Evolution. Most functions are modified by a low number of authors.

Second, we conceived a metric we call *function territoriality*. The function territoriality of author a in function f is the defined as the proportion of lines of code of f authored by a. Thus if a function is only authored by one person, such person has a 100% territoriality of that function.

Tables III and IV present the results for Apache and Evolution on a biennial basis (for all years, February 28th). As it can be observed, in both cases there is large proportion of functions that contain lines exclusively of the primary author (this is not surprising, as we have already shown that many functions are never changed). On the other hand, only one sixth of the functions in Apache, and values that range from 4% to 8% in the recent years in Evolution, are for functions where the primary author has contributed less than 50% of the lines of code.

From the figures, we can observe that the territoriality of functions in Apache is lower than in Evolution, as there is a higher number of functions where the proportion of code by the function's primary author is lower. Also, for both cases, as time passes, function territoriality becomes less marked and the contribution of the primary author drops. Both case studies are long-lived software projects and it is known that many developers have left the project [4], this may be the effect of maintenance tasks taken over by other developers and consequently making the contribution of primary authors shrink over time (and in some cases, over time, the primary author of a function changes).

Observation #3: Most functions are authored by very few persons.

D. Hypothesis #4: Orphan functions are less likely to be modified

In hypothesis #3 we have observed that there exists high territoriality of functions. At the same time we have observed that as time passes, territoriality lessens. We know that there is a high turnover in free software projects [4], which implies that the maintenance of functions has to be done by other developers.

To test this hypothesis, we need to compare the number of changes to orphan and non-orphan functions. Ideally, we would divide all functions into these two sets, being the orphan functions those whose primary author performed a last commit more than 36 months ago and the non-orphan functions the rest. But to be on the safe side, we have used smaller values of time for non-orphan functions in our study as we thought that considering a developer who left the project 35 months ago as still active is counterintuitive.

For our comparison, we take all functions of a given age and will identify those orphan and non-orphan, using two values in each case (more than 36 and 48 months for orphan functions, less than 12 and 18 months for non-orphan). For both sets of functions we compute the number of changes on them in the last 36 months. We will then compare the results between both sets. If our assumption is correct, the number of changes to orphan functions should be less than the number of changes to non-orphan functions.

Year	# Funcs	0% to 24%	25% to 49%	50% to 74%	75% to 99%	100%
2008	2163	4 (0.18%)	342 (15.81%)	600 (27.74%)	410 (18.96%)	807 (37.31%)
2006	2160	4 (0.19%)	340 (15.74%)	597 (27.64%)	414 (19.17%)	805 (37.27%)
2004	2152	4 (0.19%)	339 (15.75%)	596 (27.7%)	405 (18.82%)	808 (37.55%)
2002	2044	5 (0.24%)	320 (15.66%)	587 (28.72%)	391 (19.13%)	741 (36.25%)
2000	1907	5 (0.26%)	328 (17.2%)	560 (29.37%)	333 (17.46%)	681 (35.71%)
1998	1310	0 (0.0%)	163 (12.44%)	548 (41.83%)	249 (19.01%)	350 (26.72%)
1996	648	0 (0.0%)	0 (0.0%)	28 (4.32%)	84 (12.96%)	536 (82.72%)

Table III
APACHE. NUMBER OF FUNCTIONS (AND SHARE) BY FUNCTION TERRITORIALITY OF THE PRIMARY AUTHOR.

Year	# Funcs	0% to 24%	25% to 49%	50% to 74%	75% to 99%	100%
2008	12601	16 (0.13%)	1045 (8.29%)	3279 (26.02%)	4107 (32.59%)	4154 (32.97%)
2006	12553	4 (0.03%)	609 (4.85%)	2532 (20.17%)	2957 (23.56%)	6451 (51.39%)
2004	13933	2 (0.01%)	583 (4.18%)	3022 (21.69%)	3348 (24.03%)	6978 (50.08%)
2002	14548	0 (0.0%)	353 (2.43%)	2119 (14.57%)	3105 (21.34%)	8971 (61.66%)
2000	5415	0 (0.0%)	4 (0.07%)	263 (4.86%)	612 (11.3%)	4536 (83.77%)
1998	54	0 (0.0%)	0 (0.0%)	2 (3.7%)	2 (3.7%)	50 (92.59%)

Table IV
EVOLUTION. NUMBER OF FUNCTIONS (AND SHARE) BY FUNCTION TERRITORIALITY OF THE PRIMARY AUTHOR.

Age of Functions	Type	Primary Author Inactivity	# Functions	# Changes	Mean	Median	Mode	St Dev
$>$ 36 months	Orphan	$>$ 36 months	1569	25	0.02	0	0 (1547)	0.14
		$>$ 48 months	1410	23	0.02	0	0 (1390)	0.14
	Non-Orphan	$<$ 12 months	106	9	0.08	0	0 (98)	0.31
		$<$ 18 months	106	9	0.08	0	0 (98)	0.31
$>$ 48 months	Orphan	$>$ 36 months	1569	25	0.02	0	0 (1547)	0.14
		$>$ 48 months	1410	23	0.02	0	0 (1390)	0.14
	Non-Orphan	$<$ 12 months	81	8	0.10	0	0 (74)	0.34
		$<$ 18 months	81	8	0.10	0	0 (74)	0.34
$>$ 60 months	Orphan	$>$ 36 months	1563	24	0.02	0	0 (1542)	0.14
		$>$ 48 months	1410	23	0.02	0	0 (1390)	0.14
	Non-Orphan	$<$ 12 months	80	8	0.10	0	0 (73)	0.34
		$<$ 18 months	80	8	0.10	0	0 (73)	0.34
$>$ 72 months	Orphan	$>$ 36 months	1560	24	0.02	0	0 (1539)	0.14
		$>$ 48 months	1408	23	0.02	0	0 (1388)	0.14
	Non-Orphan	$<$ 12 months	64	8	0.13	0	0 (57)	0.38
		$<$ 18 months	64	8	0.13	0	0 (57)	0.38
$>$ 84 months	Orphan	$>$ 36 months	1510	24	0.02	0	0 (1489)	0.14
		$>$ 48 months	1380	23	0.02	0	0 (1360)	0.14
	Non-Orphan	$<$ 12 months	54	7	0.13	0	0 (48)	0.39
		$<$ 18 months	54	7	0.13	0	0 (48)	0.39

Table V
CHANGES TO ORPHAN VS NON-ORPHAN FUNCTIONS FOR APACHE. SEVERAL VALUES FOR MONTHS OF PRIMARY AUTHOR (OWNER) INACTIVITY ARE SHOWN FOR ORPHAN AND NON-ORPHAN FUNCTIONS FOR FUNCTIONS OF THE SAME AGE.

Results obtained are shown in Table V for Apache and Table VI for Evolution. As already known from hypotheses #1 and #2, the number of changes to functions is low, especially as functions age, so low numbers are expected. We can see from both case studies that for all the values (but one) the mean number of changes to orphan functions is lower than to non-orphan functions for functions of the same age. The rest of statistical measures (median, mode and standard deviation) support the evidence that orphan functions are less likely to be modified than non-orphan functions, even for the only exceptional case in Evolution where the mean of orphan and non-orphan functions is similar. With the evidence presented we can conclude that for these projects:

Observation #4: Non-orphan functions are changed more often than orphan functions of their same age.

VI. DISCUSSION

In the two projects under observation most functions never change (or change very few times). This is perhaps because most functions tend to implement a very specific functionality and do not need to change or evolve once that functionality is completed. 33% of the *Selected* functions

Age of Functions	Type	Primary Author Inactivity	# Functions	# Changes	Mean	Median	Mode	St Dev
$>$ 36 months	Orphan	$>$ 36 months	915	595	0.65	0	0 (616)	1.37
		$>$ 48 months	431	346	0.80	0	0 (266)	1.6
	Non-Orphan	$<$ 12 months	41	58	1.41	1	0 (17)	2.25
		$<$ 18 months	49	73	1.49	1	0 (18)	2.12
$>$ 48 months	Orphan	$>$ 36 months	771	477	0.62	0	0 (523)	1.35
		$>$ 48 months	431	346	0.80	0	0 (266)	1.6
	Non-Orphan	$<$ 12 months	27	44	1.63	1	1 (10)	2.2
		$<$ 18 months	34	59	1.74	1	1 (13)	2.03
$>$ 60 months	Orphan	$>$ 36 months	341	217	0.64	0	0 (228)	1.44
		$>$ 48 months	246	188	0.76	0	0 (156)	1.62
	Non-Orphan	$<$ 12 months	6	5	0.83	1	1 (3)	0.75
		$<$ 18 months	10	12	1.20	1	1 (5)	0.92
$>$ 72 months	Orphan	$>$ 36 months	254	174	0.69	0	0 (159)	1.42
		$>$ 48 months	179	153	0.85	0	0 (103)	1.64
	Non-Orphan	$<$ 12 months	6	5	0.83	1	1 (3)	0.75
		$<$ 18 months	9	11	1.22	1	1 (4)	0.97
$>$ 84 months	Orphan	$>$ 36 months	120	110	0.92	0	0 (64)	1.69
		$>$ 48 months	98	104	1.06	1	0 (48)	1.83
	Non-Orphan	$<$ 12 months	2	3	1.50	1	1 (1)	0.71
		$<$ 18 months	5	9	1.80	2	1 (2)	0.84

Table VI
CHANGES TO ORPHAN VS NON-ORPHAN FUNCTIONS FOR EVOLUTION. SEVERAL VALUES FOR MONTHS OF PRIMARY AUTHOR INACTIVITY ARE SHOWN FOR ORPHAN AND NON-ORPHAN FUNCTIONS FOR FUNCTIONS OF THE SAME AGE.

in Apache (and 61% in Evolution) never changed once they were created. We suggest further research on the low number of changes to functions as we don't know if this is because of good quality control from their authors (functions are tested well before they are added) or because many of these functions are dead-code; further dynamic and static analysis is needed to determine this.

Of the functions that change, does it matter if the author is still within the project or not? We found that, for these two projects the answer is yes. Perhaps it is the reticence of developers to modify somebody else's code, or the lack of understanding of what the code of somebody who is no longer within the project does. This triggers important questions: is it more productive to ignore the orphan code, or it is more productive to keep modifying it? How can we reduce the impact of authors departing a project? This is an issue that is even stronger in industry, where personal rotation is always present.

Some more consequences may be drawn from evidence #4. First, it is a demonstration that the evolution of software also depends on the original developers staying in the project. Even if the software code base has not suffered changes, a developer abandoning the project affects the project. Second, we may infer that orphan functions are changed less often because they are more difficult to change and hence more difficult to maintain. If this is true, developer turnover could be another hint for software decay (which underlies the assumption that a unit of code is decayed if it is *harder to change than it should be*, measured in terms of effort, interval and quality) as stated by Eick *et al.* [5] or in the sense of *software aging* by Parnas ("programs lose their appeal and that maintenance becomes a burden, the same for good and bad programs") [6].

All the results shown in this paper let us think that functions may better serve as the minimal structure for the study of software evolution, in detriment of other granularities usually used as files or SLOC. As we have seen, they could be considered as the atoms, the basic unit, of software evolution. They have special characteristics that are of great advantage for this: they have semantic meaning, many seldom change, and they are closely related to their authors.

VII. THREATS TO VALIDITY

In terms of internal validity, the biggest threat to our study is that we track names of functions, and we only track them within the main trunk of the version control system of the project. We do not track the history of a function outside the version control system (if a function has a long evolution before it is inserted into the project – for example, the initial code of Apache 1.3 is inherited from version 1.2). If a function is renamed or relocated we considered the original function as deleted and the new name as a new one. To minimize such effects we have selected a subset of the functions (we have called this set *Selected* functions). As described in III-A, this set is expected to contain functions with their entire history. We were surprised that most observations are very similar between *Selected* functions and all functions in both projects.

Another factor that could affect internal validity is code reformatting. While tracking the evolution functions we have made sure that we ignore any changes in whitespace and reformatting. We have observed that, except for comments,

there are very few changes that do not affect source code and show that most code is own by the person who is responsible for it.

With regard to external validity we have chosen two projects with rich history, but very different goals and domains. One is a relatively small server application (Apache), the other is a very complex and large mail client with a very rich user interface (Evolution). In both cases the results we observed are similar, giving us confidence that the results herein presented will be observed in other open source projects. Evolution has one peculiarity: it has been a project where most of the development has been sponsored by one organization (originally called Helix, renamed to Ximian, and latter acquired by Novell—its current owner). Even though it is an open source project, most of its lines of code come from employees of its owner company (see [7]). It is very likely that what we have observed in Evolution will also be observed in commercial software. It is, nonetheless, important to replicate this study with other projects, both in industry and open source.

A. Reproducibility of the Study

According to the reproducibility classification criteria proposed in [8], the attributes of this study are given in Table VII. Detailed information can be obtained at http://gsyc.urjc.es/˜grex/wetsom2012.

VIII. Conclusions

The main contributions of this paper are threefold. First, we have proposed metrics for the evolution of functions based on the number of times they have been modified, and who make the modifications. Second, we have shown that analyzing the evolution of software at the semantic (function) level instead of at the physical (file, line) level allows to gain a lot of understanding and new knowledge on the software project under study. We have shown that most functions rarely change, and if they do, they usually change early in their lifes. And third, we have observed that the change pattern of functions depends heavily on the continuous participation of its creator in the software project.

Element	Assessment	Condensed Assessment
Data source	usable	U
Retrieval methodology	usable	U
Raw dataset	usable	U
Extraction methodology	usable likely available in future flexible	U + *
Study parameters	Usable	U
Processed dataset	Usable likely available in future flexible	U + *
Analysis methodology	Usable	U
Results dataset	Usable	U

Table VII
Reproducibility assessment of this study.

While the first two contributions can be seen as a technical improvement to technical-related issues of software evolution, the third one goes beyond and illustrates the importance of the *human factor* in the field of software evolution and maintenance, which in the opinion of the authors is an area of software metrics that has not been sufficiently studied until know. Future work should continue, with special attention to how this affects software maintenance, especially in relation to software decay and/or software aging.

Acknowledgments

The work of G. Robles and D. Izquierdo-Cortázar has been funded in part by the European Commission under project ALERT (FP7-IST-25809) and by the Spanish Gov. under project SobreSale (TIN2011-28110).

References

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

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

[3] D. M. Germán, "Using software trails to reconstruct the evolution of software," *J. of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 6, pp. 367–384, 2004.

[4] G. Robles and J. M. González-Barahona, "Contributor turnover in libre software projects," in *OSS*, 2006, pp. 273–286.

[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[6] D. L. Parnas, "Software aging," in *Proceedings of the International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 279–287.

[7] D. M. Germán, "An empirical study of fine-grained software modifications," in *Proc Intl Conference in Software Maintenance*, Chicago, IL, USA, 2004.

[8] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *Empirical Software Engineering*, vol. 17, no. 1-2, pp. 75–89, 2012.