Evolution of understandability in OSS projects

Andrea Capiluppi Maurizio Morisio Patricia Lago

{andrea.capiluppi, maurizio.morisio, patricia.lago}@polito.it

DIP. AUTOMATICA E INFORMATICA

POLITECNICO DI TORINO

ITALY

ABSTRACT

Empirical papers on Open Source software should try and formulate reasons for successes as Linux, Apache and some other flagship projects. What we need to understand about this topic is on the process of producing software through cooperation of different efforts. Albeit many success reasons for these projects are inherently due to the application domain that the project develops, architectural and conceptual views of the code have to be considered as key factors when considering community efforts and joint decisions.

In this work we focus our attention on what is perceived of a source code when investigating its structure. We do this considering that structure as a proxy for the conceptual architecture of the application. A metric is developed based on some current assumptions, and it is tested over a sample of Open Source projects. What is interesting to note, is that refactoring efforts are clearly visible when intended as reduction of complexity of source code. Our second observation is that, based on what an Open Source software currently does, i.e. its application domain, there's a threshold value that several projects tend to.

1 INTRODUCTION AND RELATED WORK

Empirical works on software engineering should try and describe interesting features in the processes and products of real systems (E-type software, as described in [11]). Empirical research on Open Source[1] software (OSS) should at first define if those have to be considered E-type applications: flagship projects (e.g. Linux, Apache, etc) are not only jointly developed and tested, but are also widely used in real situations. Next, empirical OSS studies have to be performed to assess and test established theories, or to define new ones.

The first aspect that drives our research is using publicly available data to perform measures related to Open Source systems. In a previous study [1] we discovered that given a large sample of OS projects, very few can count on several (i.e. more than one) developers. In a further work [2] we studied what happens in those projects where enough developers join together in a common effort. However, we couldn't understand (nor assess) why certain projects could resolve in a faster evolution, in a larger growth of functionalities, or even in a larger number of developers. What's more, we observe that these attributes don't correlate with each other, and a large pool of developers is not developing the largest sized project. In general, a good number of recent studies ([1], [2], [3], [4], [5]. [6], [7] and [8]) have been involved in life-cycle analyses of Open Source projects. What's actually needed is a common ground for comparing these two processes, with a rigorous approach that the 'Cathedral & Bazaar' terms and established definitions are still missing.

The second element which is driving our work comes from first and introductory studies on OSS [7]: there's, a common assumption of Open Source effort as the joined struggle of thousand developers, easily collaborating through the web with highly efficient communication tools. The GNU/Linux kernel has contributed to create such an image for a typical Open Source project. This view is as typically shared, as far from reality. Recent studies have normalized this common belief, and what is true for most established projects is no more adapted to any other OSS project. Few are the active developers, less are the occasional developers, and even the beta-tester pool is not based on tens of thousand individuals, but rather less.

Based on this, we believe that understandability is a key aspect for maintainability and for team work.

Our assumption then is as follows: OSS understandability behaves no differently from 'normal' source code comprehension. We then formulate our research questions as follows: how understandability evolves in time for OSS projects?

We draw our hypotheses from empirical Lehman's laws on software engineering: since software size evolves in time increasing (first law), and since code refactoring is necessary to prevent excessive code complexity, we suspect that understandability follows the same pattern and decreases as time passes by.

First results of this paper show that there is a general increase in understandability of source code as the project grows, and that there is a threshold value that the OS projects go towards.

2 RATIONALE

Let us first recall some initial results we got with a sample of OSS projects in a previous work [2]: therefore, a brief summary is needed. Based on a larger sample of 400 projects, we extrapolated 12 projects to be analyzed further: this selection was made based on what projects had the larger number of developers. What we observed in the conclusions was as follows: there's no correlation between a project's size and its life-span, and we can't pull any dependency-relationship between active developers and again the project's size. Smaller

1 There's some debate among communities about terms describing software released through open licenses (GPL and the such). Free Software and Open Source Software share many technical commonalities, but are quite different from the communities' point of view (see for example [16]). In this study we consider Open Source and Free Software as synonyms.

Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04)
1534-5351/04 $ 20.00 © 2004 IEEE

projects are generally moving faster in their evolution, but this is normal in any software environment: large projects are generally older and they can count on a larger number of developers. When analyzing code structure, we observed both files and directories which compose the overall source code: we found different kinds of modularization, but no clear correlation between either size and modules, or developers and modules.

Having said that, what is driving our attention for this research is perceptible in two figures that we plot below. Figure 1 is showing total number, and average size, of modules (intended as directories) in a medium project (Weasel project). Figure 2 is depicting the same plot for a larger project (ARLA project).

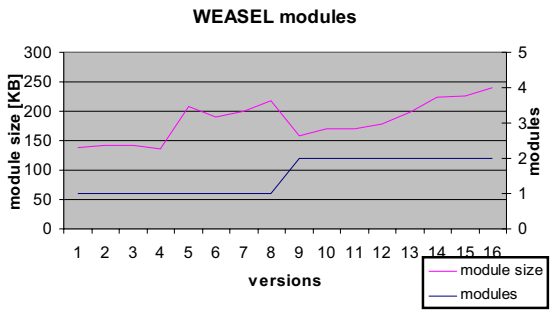

Figure 1 Weasel-project's evolution of modules size

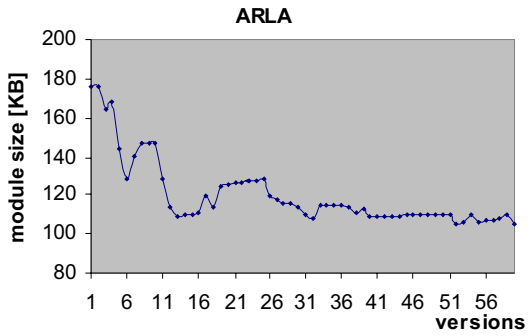

Figure 2 ARLA-project's evolution of modules size

What is interesting to us is observing how these two projects have evolved their average size of what we called modules of an application, i.e. the project's directories containing source code. In the case of Figure 1, we observe a discontinuity in version (i.e. release) 9, that we interpret as a process of refactoring due to the reorganization of source code in different directories. In the case of Figure 2, on the contrary, the process of reorganization is spanned all over the lifecycle of the ARLA project and no discontinuities are evident. In other cases, though, the project's modules tend to increase their size all over the life-cycle of a project: intuitively, that means that there is more source code in a directory, and this can be a first index of an increased 'complexity'.

Based on the studies performed in [9], when a regular trend of software evolution shows discontinuities, there is a need for recalibrating models for evolution. We therefore are interested in understanding under which conditions these discontinuities are driving easier or more difficult comprehension of the code.

Using a GQM approach, we formulate our work as follows:

Goal: Analyze decay and refactoring of OSS projects to characterize the process, from the point of view of software developers and maintainers

Questions: How size and complexity of OSS projects evolve over time? When and why discontinuities arise?

Metrics: size, number of modules, size of modules, average size of module.

3 METHODOLOGY

Based on the same sample of 400 projects, for this study we chose the OSS projects that have (at present time) the bigger size among others: 19 projects are therefore selected and analyzed further.

We then tried and collected the major number of available releases (see below for this attribute's definition) from those projects, where not all of them were in place. We in fact noticed that for bigger and older projects, it's extremely difficult to track the entire history of evolution: only latest and stable releases are currently available.

We later on used some shell scripts to parse data in usable formats, and extracted similar attributes for all projects.

Finally we formulated an indicator of understandability which we used to decide if and where a certain project was going towards a larger number of files, in a minor number of macro-folders.

In Section 4 we formulate our definitions of attributes; in Section 5 we show some results of these attributes' observation. Section 6 is dedicated to understand if and where our complexity metrics succeeds (or fails) in describing a project's overall behavior. Finally, Section 6 suggests some conclusions and future work.

4 ATTRIBUTE DEFINITION

Being in a metrical context, we have to formulate several metrics for what we are studying. Each attribute we present is followed by a short description.

4.1 Releases

A release of a software project is intended as the moment when the project team makes the source code openly available for public inspections. That can happen after small overall fixes, or after pulling major changes or new functionalities in the project. There is a common assumption on numbering conventions: even numbers should declare a global stability and thorough testing of the code functionalities. Odd numbers instead are considered instable phases in code evolution.

4.2 Code size

This metric is not necessary for an absolute comparison with other OSS projects. We need it instead to give an idea of what is the relative change over the different releases of a project's life-cycle. With simple scripts, we evaluated both the bytes (and KB) of each file composing the source code of an application, and their LOCs (and KLOCs). For each project, a 'map' of source files was derived (e.g. '*.c' meaning all files whose extension is 'c', typically written in C code). That map was looked after for all releases of the same project, and the results were parsed to give a simple pair-structure of (release_nr, code_size), which was finally plotted.

4.3 Macro-modules

As a first index of a project's modules, i.e. the first-order groupings of source code, we believe that directories can play a role in understanding the conceptual architecture of an application. We are therefore interested in evaluating which, and how many, directories contain code in each version for each project. We have also to isolate these from the directories not containing any code, but simply auxiliary files (as the documentation, images, etc.).

When approaching complexity, and in particular the next discussed 'perceived' complexity, there is a dual aspect to be considered: one regards directories (or macro-modules), the other regards files (or micro-modules). The more numerous directories, the more complex the code is perceived. The less dispersed the directories (many files in each of them), the more complex the code is perceived.

4.4 Micro-modules

If we consider the LOCs (lines of code) as the simplest measure for change, and the directories as the macro-modules for determining how code is structured, we can consider files as intermediate metrics. We call files as micro-modules, and we are interested in how they are added or deleted in an OSS project. As for macro-modules, we tracked their number and relative size for every release of a project, for all the projects. When considering the perceived complexity, we assume that the bigger the file, the more complex the project is perceived.

4.5 Application Domain

We are interested in understanding if there's a relationship between complexity and what a project is being developed for. This attribute could be biased by our understanding of the software application domains. That's why we used the FreshMeat portal for determining the different application domains for our projects: considerations on the correlation between trend of complexity and domains are finally given.

4.6 Delta size

As well as code size, for providing insights on the different projects, we calculated also the relative change of code (measured in KB) from the first recorded release to the last available.

4.7 Understandability measure

We hypothesize that two factors influence the a developer's understandability of a system: the dispersion of macro-modules, and the relative size of micro-modules. We define dispersion as the percentage of micro-modules contained in every macro-module: each macro-module has at least one file, for its definition of directory containing source code. Each macro-module can have an unlimited number of files, which determines its dispersion. We therefore define the code indistinctness as follows:

$$CI = A \times B \text{ where}$$

Equation 1 – definition of code indistinctness

$$A = \left(1 - \frac{Nr.ofMacroModules}{Nr.ofMicroModules}\right)_{forEveryVersion}$$

Equation 2 – contents of macro-modules

$$B = \left(\frac{SizeofApplication}{Nr.ofMicroModules}\right)_{forEveryVersion}$$

Equation 3 – average size of micro-modules

Equation 1 is expressed in the same metrics of the B term, and it can be LOCs, KBs, etc. This expression therefore shows how much of the application size is 'obfuscated' through structures as micro- and macro-modules. If there's a single macro-module, called say 'network', which contains one micro-module, named say 'socket.c', the code indistinctness is zero. Or, it's easy to understand what the micro-module means inside of the macro-module. In the following we will consider 'code indistinctness' and 'understandability' as antonyms.

Equation 2 can be intended as 'the percentage of code which is not explained by macro-modules': if a project contains 10 macro-modules and 100 micro-modules, we can say that these 10 micro-modules out of 100 are directly understood as they are included in a particular macro-module. The other 90 are not directly explained, since they are included with others in a macro-module. We therefore can formulate Equation 2 as a metric for understanding how macro-modules are filled with source code files. The term A ranges between [0,1], since there's at least one micro-module per macro-module (i.e. there's at least one file per directory containing code). As long as a single macro-module increases the number of micro-modules contained, the term A gets larger. If each macro-module contains exactly one micro-module, the term A goes to zero.

The B term in Equation 3 is the average size of micro-modules for each release: the idea here is that the larger the file, the lower is its comprehensibility. Term B alone is giving a partial view of the same problem: it explains how the average size [LOC, KB, etc.] of micro-modules has evolved over time in a project's perspective. The union of the A and B terms give a better view of how much source code of a project's release is dispersed over its macro- and micro-modules. We use this proxy also to understand what code structure is typical for Open Source projects, and how it changes when their size grows from small to large applications.

We are taking into account only code-structure measures, but we are well aware of the fact that code unserstandability is also a matter of personal and subjective issues. Nonetheless code measures can be automatized and can serve as a first step for

5 ATTRIBUTE ANALYSIS

5.1 Application domains

In we plot the distribution of domains in our sample: since several projects have different topics associated, we counted those projects as having more topic. A database server is typically a Database AND an Internet application: in this case we count the Database and Internet domains, albeit a single project.

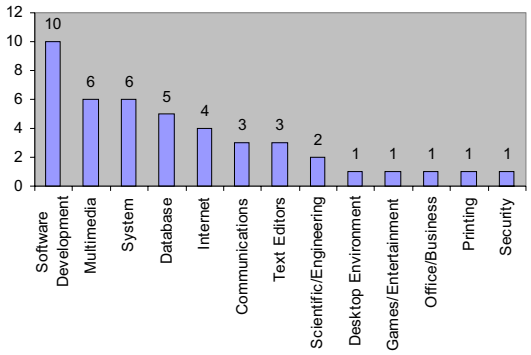

Figure 3 Distribution of application domains in the sample

It is interesting to compare horizontal applications (applications used to build other software, the end user is required to program and is, likely, a software professional) with vertical ones (applications used by an end user, no programming is required). Horizontal applications (categories Internet, System, Software development, Communications, Database, Security) account for 68%. If we observe this distribution made in [1], we observe a similar percentage of 62%. In both situations where the sample is composed of projects with more developers, and when is chosen by larger project size, the relevance of horizontal applications still describes OSS software as expert tools for expert programmers.

Table 1 summary of relevant attributes used in our analysis (initial and final refer to first and last releases available)

PROJECT	Macro-modules (initial)	Macro-modules (final)	Micro-modules (initial)	Micro-modules (final)	KLOCs (initial)	KLOCs (final)	KB (initial)	KB (final)	Delta	Application domain
GANYMEDE	3	3	26	26	218	232	77	81	**4,7%**	**H**
GHEMICAL	12	12	424	392	219	229	3800	4025	**5,9%**	**V**
GWYDION DYLAN	57	132	502	1056	218	357	6340	10877	**71,6%**	**H**
GIMPPRINT	1	14	7	158	11	94	305	2508	**723,2%**	**V**
GIST	28	38	776	1063	173	192	4055	4437	**9,4%**	**H**
GRACE	14	13	253	309	117	161	3190	4500	**41,1%**	**V**
HTDIG	64	24	602	345	157	151	5090	3229	**-36,6%**	**H**
IMLIB	5	5	21	31	51	52	2541	2545	**0,2%**	**H**
KSI	17	12	181	144	114	107	2289	2416	**5,6%**	**H**
LINUXCONF	37	97	575	1268	105	237	2562	5891	**130,0%**	**H**
LRCZO	3	10	19	240	6	111	197	3746	**1797,0%**	**H**
MIT-SCHEME	31	31	961	1139	341	428	10839	13207	**21,8%**	**H**
NICESTEP	4	16	44	122	34	75	1206	2489	**106,4%**	**H+V**
PLIANT	23	79	310	1092	65	246	1912	8165	**326,9%**	**H**
QUAKEFORGE	10	40	210	475	203	232	4276	4755	**11,2%**	**V**
RRDTOOL	12	25	86	114	87	106	1695	2009	**18,5%**	**H+V**
SIAGOFFICE	4	18	23	209	13	111	276	2549	**823,5%**	**V**
VOVIDA	1	127	49	2658	13	669	416	19876	**4675,8%**	**H**
XFCE	12	16	203	335	46	100	1318	2850	**116,3%**	**V**

5.2 Project Analyses

We analyze here more deeply some of the 19 projects' attributes: for reasons of space we depict the same trend for each project from Figure 4 to Figure 22.

5.2.1 GANYMEDE

Ganymede is a Java-based directory management software system: the FreshMeat site refers that its domains are Database and System (horizontal domain). We tracked all the releases (12) which are spanned over 5 years of lifetime. Its internal structure didn't change for all of the releases (28 micro-modules, 3 macro-modules): since the code evolved in size, we expect that our calculated measure of code indistinctness grows with a linear trend, which is depicted as the 'complexity' line in Figure 4. What is relevant to us is the range of it: in all of its history, our metrics is in a [2.6 – 2.8] KB interval.

5.2.2 GHEMICAL

Ghemical is computational chemistry software: the FreshMeat site refers that its domains is Scientific/Engineering. (vertical domain) We have access to all the releases here (6) which cover a period of 1½ years. We observed a constant growth in the project's size until the very last release, which lowers the number of micro- and the macro-modules, as well as the project's overall size. We interpret this as code and modules reorganization, in short refactoring. The trend of code indistinctness is constantly growing, and it gets from 8.7 to 10 KB. (Figure 5)

5.2.3 GIMP-PRINT

Gimp-Print is a printing utility: FreshMeat categorizes it as Multimedia application (vertical domain). We don't have access to the whole number of releases: we only observed 93 of them (3 ½ years). Delta size is one of the largest in our sample (723,2%). We are intrigued by the

trend of our measure in this project: in some 1/3 of its life-cycle, the code underwent some reorganization, and after that we observe a constant decrease of this attribute. Its peak is observed with 87 KB: the lowering trend stops at 14 KB in our last observation. Reorganization is not on number of micro-modules, which keeps on increasing. (Figure 6)

5.2.4 GIST

GIST is a tool for creating dynamic web sites: FreshMeat records its domain as Communications, Internet, Software Development and Database (horizontal application). We have access to all releases of it (20 releases, 4 years). The overall trend of micro- and macro-modules is constantly growing, albeit some size reorganization on release 5 and 8 (Figure 7). The code indistinctness measure keeps on the same levels (4 KB)

5.2.5 GWYDION-DYLAN

Gwydion-dylan is an implementation of the Dylan programming language: we don't have the whole history (15 releases). Based on this, we assume an horizontal domain. Micro- and macro-modules are constantly growing (they both double their initial size), and the code indistinctness measure keeps on lowering. We recorded the lowest point around 9 KB (Figure 8).

5.2.6 GRACE

Grace is an XY plotting tool for workstations, and FreshMeat tracks it as a vertical application (Multimedia). We don't have access to the whole history (21 releases). Micro-modules have a constant growing trend; macro-modules stabilize on a single value, while code indistinctness has a growing trend (from the initial 11 KB to the actual 14, Figure 9).

5.2.7 HT://DIG

Ht://Dig is a world-wide-web search system. Its domain is Internet (horizontal). We have access to a very limited portion of life-cycle (10 releases): this can reduce the validity of our conclusions. We observe a reduction in both the micro- and macro-modules. We also see a general stable/decreasing trend of code indistinctness, albeit release 10, on average 6.5 KB. (Figure 10)

5.2.8 IMLIB

Imlib is a general Image loading and rendering library: since it's a library, FreshMeat categorizes it as Sw Development (horizontal domain). We access to 14 releases, not the whole history. We observe a smooth growing trend in micro-modules, and some reorganization around release 8. Code indistinctness is quite high and generally constant (around 70 KB, Figure 11)

5.2.9 KSI

KSI is a Scheme implementation written in C: FreshMeat refers it as a Software Development project (horizontal). We have access to a very limited number of releases (12 versions). We observe a continue decreasing of micro- and macro-modules. In first releases code indistinctness is around 12 KB, then it jumps to 15 (Figure 12).

5.2.10 LINUXCONF

Linuxconf is a fully integrated utility for administering Linux systems. FreshMeat categorizes it as a System project (horizontal domain). We have access to nearly all its releases (52 observed): we observe a very fast evolution trend up to release 19, where a discontinuity in macro-modules is observed. Around release 47 an opposite discontinuity is observed. Understandability is very stable in the whole observation (around 4 KB, Figure 13).

5.2.11 LCRZO

Lcrzo is a network library, for easily creating network programs. FreshMeat categorizes it as a Software development project (horizontal domain). We have access to the whole history of it (56 releases, 3 years). Its components evolution is very intriguing to our study: we observe massive reorganization around release 40, which corresponds to an (isolated) peak of its code indistinctness. In general, the Lcrzo results are worse than the other observed up to here (average is 14 KB, but with a large variance, 26). After release 40, macro-modules and code indistinctness seem to stabilize (Figure 14).

5.2.12 MIT-SCHEME

Scheme is a full programming language developed at MIT Institute. FreshMeat categorizes it as a Software development project (horizontal domain). We observe only the latest 26 releases. This is the largest project in our sample: delta size is 'not-as-large' as other projects in our sample, but we believe that this is due to its initial size (some 10 MB). Macro-modules don't evolve in our observation (31). We observe reorganizations of micro-modules in releases 8 and 15. We observe also a stable factor of code indistinctness: on average, 10.9 KB with a variance of 0.13 (Figure 15).

5.2.13 NICESTEP

NiceStep is reported by FreshMeat like being a Internet, Multimedia and Software Development project (a mix of horizontal and vertical domains). It's one of the (initially) smallest project of our sample, and it gets visible changes in its releases, which we report completely. Its macro-modules gets 4 times as big; its micro-modules three times. Its code indistinctness is globally decreasing, and it gets the lowest point in 17.5 KB. There's no visible stabilization point for it (Figure 16).

5.2.14 PLIANT

Pliant is a complete programming language framework, which can be as light as an interpreted language, and as full as a linker language between different languages components. FreshMeat reports it as being a Communications, Internet and Software Development project (horizontal domain). We can't track its entire history, but some half of it (36 versions, most of them recent ones). Its code size is constantly growing, its macro-modules get four times as big, its micro-modules three times. We observe a stabilization point for code indistinctness, around an average of 6.1 KB (variance is 0.18, Figure 17).

5.2.15 QUAKEFORGE

Quakeforge is a commercial game by Id Software which was released as an open source application in 1999. An OS team then gathered on it, and it is now developing it further. We consider it a vertical application, as FreshMeat stores it as Games/Entertainment and Multimedia project. We have tracked its entire history, which consists of 17 releases. We observe a massive code evolution on version 15, as well as modules reorganization. There's no clear stabilization point for

code indistinctness, albeit a decrease of it is visible (Figure 18).

5.2.16 RRDTOOL

RRDTool is a tool for logging and analyzing data gathered from different kinds of data-sources. FreshMeat categorizes it as a Multimedia, Software Development and Database project (both vertical and horizontal domain). We can record only a part of this project's releases (16), mostly newer releases. The discontinuity point in release 2 is due to unavailability of intermediary data. If we consider releases from that point on, we observe a smooth evolution trend of micro-, macro-modules and size. Code indistinctness measure stabilizes around 13.7 KB (Figure 19)

5.2.17 SIAG-OFFICE

SIAG-Office is an office application which comprises a spreadsheet, a databases handler and a word processor. We categorize it as a vertical application (FreshMeat stores it as an Office/ Business project). We can observe only a part of the project's releases, the earliest ones are unavailable. Its evolution trend is quite regular for all attributes plotted, but there's a discontinuity on release 32, which reduces size and micro-modules. Macro-modules remain unchanged. There's no clear stabilization point for code indistinctness (Figure 20).

5.2.18 VOVIDA SIP STACK

VOVIDA SIP Stack is the OS implementation for the 'voice-over-IP' stack. We have data for all its releases (15). FreshMeat reports it as a joint Communications and System project (horizontal topic). Size, micro- and macro-modules evolve constantly along the life-cycle of this application. We find a stabilization point for code indistinctness around 7.3 KB (with a variance of 2.1 KB, Figure 21).

5.2.19 XFCE

XFCE provides a graphical environment and interface for Unix based systems. We consider it a vertical application since FreshMeat stores it as a Desktop Environment project. We can't track all its releases, only the most recent ones. Macro-modules have a smoother trend than micro-modules, except a discontinuity point in release 30. The code indistinctness is quite stable, albeit increasing, over some 7.15 KB (Figure 22).

6 CONCLUSIONS, VALIDITY THREATS AND FUTURE WORK

We have presented here a thorough analysis of 19 OSS projects: for each of those a set of attributes was calculated, for each release. We also have defined and computed code indistinctness, an indicator for code understandability in OSS projects..

Discontinuity points in code indistinctness are in general more frequent and steep for vertical, or joint horizontal-vertical domains applications.

Nearly all projects here examined tend to increase their understandability, if it's intended as organization of code in micro- and macro-modules. Vertical applications have in general higher values as stabilization points (if any). Horizontal applications have a stabilization point in the range [5-10] KB. Table 2 shows this trend: when a Y/N appears, it refers to recognized efforts in any phase of a projects' evolution, not getting a stable, final result (see HTDIG and PLIANT projects, Figure 10 and Figure 17).

We believe that analysis of micro- and macro-modules gives a better understanding of a code conceptual architecture, which is one of the drivers for some OSS successes. When those modules are structured in a comprehensible way, the benefit for a project is similar to the improved perception of a piece of code with intuitive data structures, or insightful method names.

Certainly our formulation is not explaining at a sustainable level the presence of outliers within our data set, albeit through some application domains reasoning.

Future work will be performed analyzing how this simplified vision of code architecture can actually match with more complicated ones (structure graphs, workflow of developer efforts), as well as to test if understandability can play a role in attracting new developers into an Open Source project.

Table 2 - Summary on efforts of improving understandability

PROJECT	APPL. DOMAIN	EFFORTS FOR UNDER-STANDABILITY
GANYMEDE	H	***N***
GHEMICAL	V	***N***
GWYDION DYLAN	H	Yes
GIMPPRINT	V	Yes
GIST	H	Yes
GRACE	V	***N***
HTDIG	H	Y/N
IMLIB	H	Yes
KSI	H	***N***
LINUXCONF	H	Yes
LRCZO	H	Yes
MIT-SCHEME	H	Yes
NICESTEP	H+V	Yes
PLIANT	H	Y/N
QUAKEFORGE	V	Yes
RRDTOOL	H+V	Yes
SIAGOFFICE	V	***N***
VOVIDA	H	Yes
XFCE	V	Yes

7 REFERENCES

[1] Capiluppi A., Lago P., Morisio M., 2003, Characteristics of Open Source Projects. In the Proceedings of the 7th European Conference on Software Maintenance and Reengineering, March 2003

[2] Capiluppi A., 2003, Models for the evolution of OS projects. In Proceedings of the 2003 International Conference on Software Maintenance, Amsterdam, the Netherlands, September 2003.

[3] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In Proceedings of the 2000 International. Conference on Software Maintenance, San Jose, California, Oct. 2000.

[4] Y. Ye, K. Kishida, K. Nakakoji, Y. Yamamoto, A. Aoki, 2002. Creating and Maintaining Sustainable Open Source Software Communities. In Proceedings of International Symposium on Future Software Technology (ISFST2002)

[5] S. Krishnamurthy, 2002, Cave or Community?: An Empirical Examination of 100 Mature Open Source Projects. In First Monday, Vol. 7, No. 6, http://firstmonday.org/issues/issue7_6/krishnamurthy/index.html

[6] G. Robles-Martínez, J. M. González-Barahona, J. Centeno-González, V. Matellán-Olivera, L. Rodero-Merino, 2003, Studying the evolution of libre software projects using publicly available data. In Proceedings of 3rd Workshop on Open Source Software Engineering at the 25th International Conference on Software Engineering. May 2003

[7] E. Raymond, 1998, The Cathedral and the Bazaar, FirstMonday, Vol.3 N.3 - March 2nd. 1998, on line at http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

[8] A. Mockus, R.T. Fielding, J.D. Herbsleb, 2002, Two Case Studies of Open Source Development: Apache and Mozilla. In ACM Transactions on Software Engineering and Methodology Vol.11, No. 3, 2002, 309-346.

[9] J. Ramil, 2003, Continual Resource Estimation for Evolving Software. In Proceedings of the 2003 International Conference on Software Maintenance, Amsterdam, the Netherlands, September 2003.

[10] FreshMeat OSS portal, http://freshmeat.net

[11] M.M. Lehman, J.Ramil, 2002, Software Evolution and Software Evolution Processes, Annals Of Software Engineering, Vol. 14, pp. 275-309

[12] M. M. Lehman, L. Belady, 1985,. Software Evolution, Academic Press, London, p. 33-200

[13] M. M. Lehman, 1980, On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle, J. of Sys. and Software, 1, (3), pp. 213-221

[14] M.M. Lehman, J.Ramil, 2000, Towards a Theory of Software Evolution and its Practical Impact. Proceedings Intl. Symposium on Principles of Softw. Evolution, ISPSE 2000, 1-2 Nov, Kanazawa, Japan, pp. 2 – 11

[15] A. Fenton, 1994, Software Measurement: A Necessary Scientific Basis. On IEEE Transactions on Software Engineering, Vol.20, No.3, Pagg. 199-206

[16] Why ``Free Software" is better than ``Open Source", http://www.gnu.org/philosophy/free-software-for-freedom.html

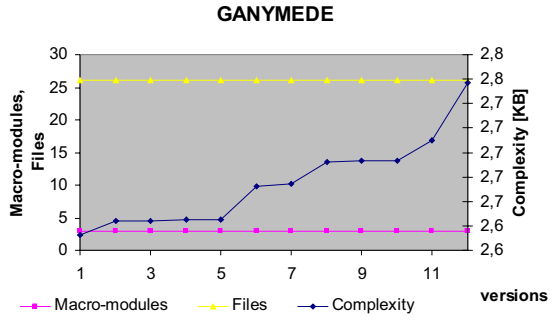

Figure 4 Ganymede project's observations

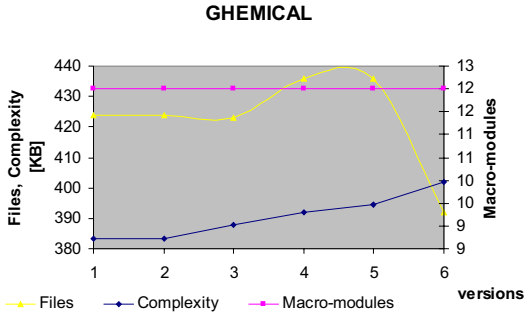

Figure 5 Ghemical project's observations

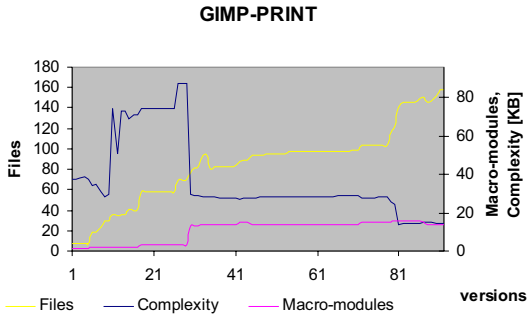

Figure 6 Gimp-Print project's observations

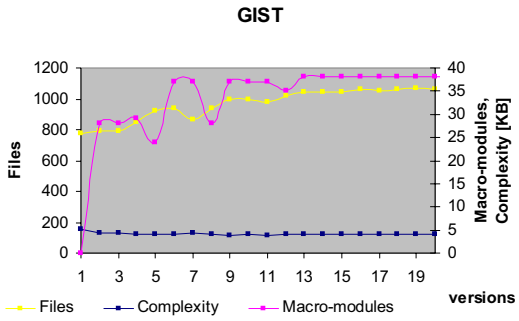

Figure 7 Gist project's observations

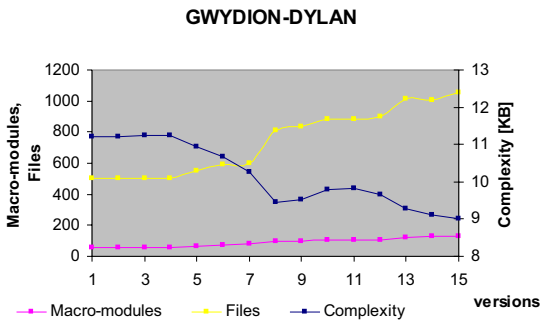

Figure 8 Gwydion-Dylan project's observations

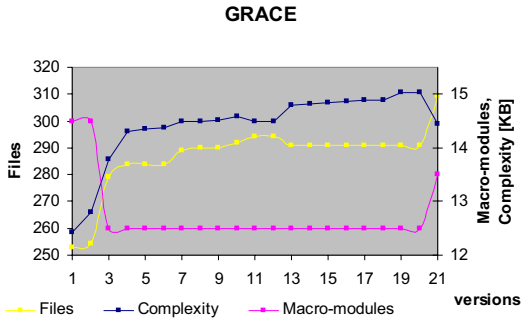

Figure 9 Grace project's observations

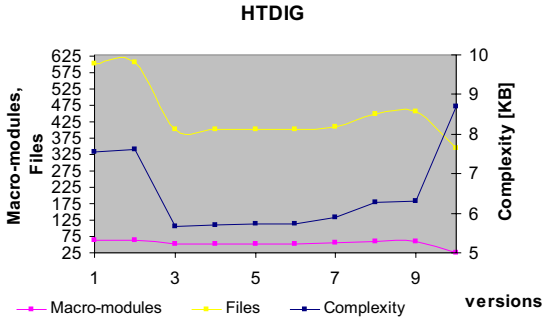

Figure 10 Ht://Dig project's observations

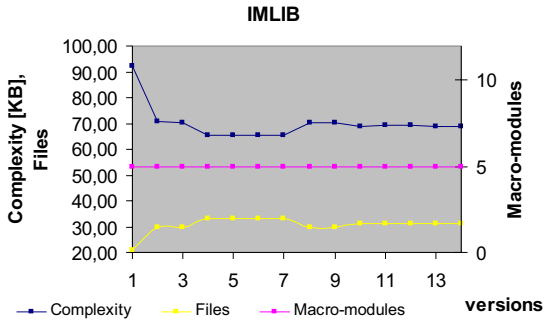

Figure 11 IMLIB project's observations

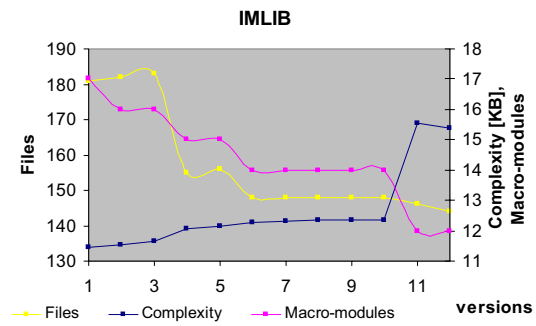

Figure 12 KSI project's observations

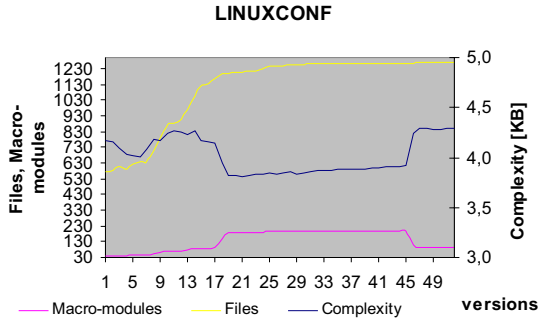

Figure 13 Linuxconf project's observations

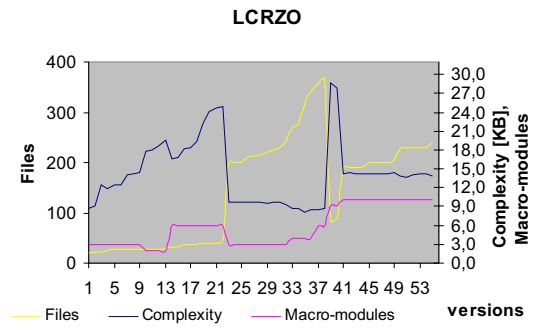

Figure 14 LCRZO project's observations

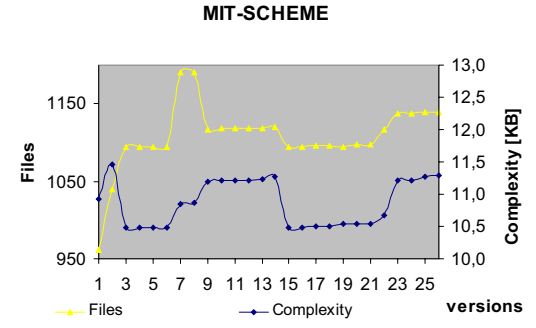

Figure 15 MIT-Scheme project's observations

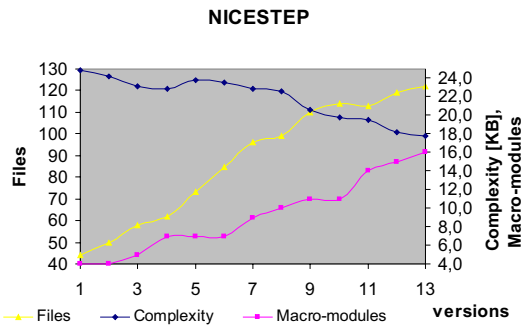

Figure 16 NiceStep project's observations

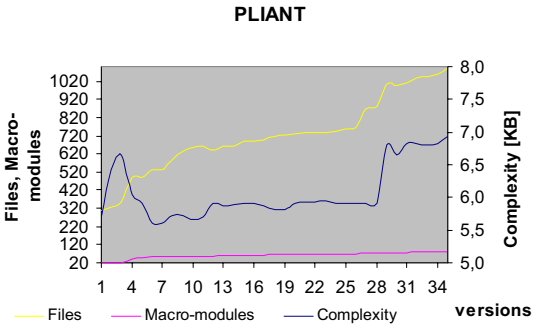

Figure 17 PLIANT project's observations

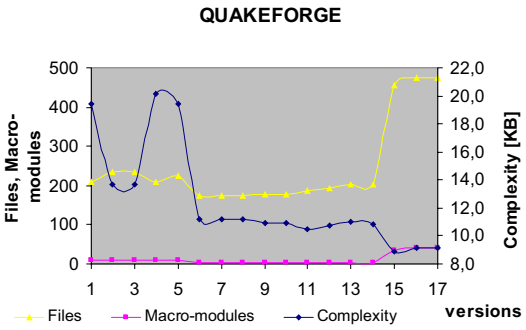

Figure 18 QuakeForge project's observations

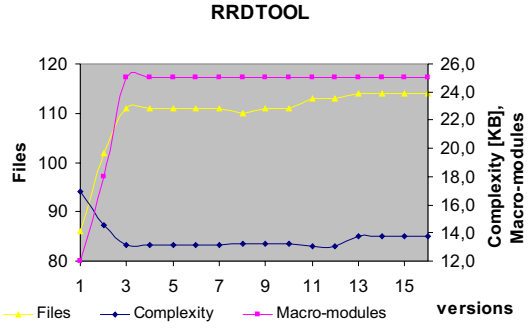

Figure 19 RRDTool project's observations

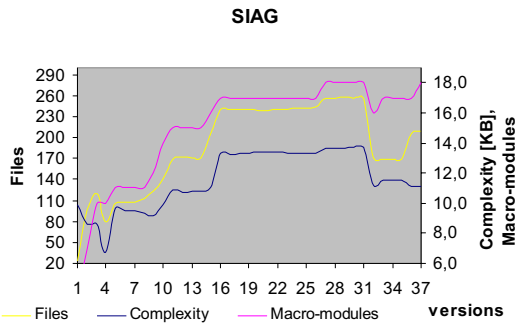

Figure 20 SIAG project's observations

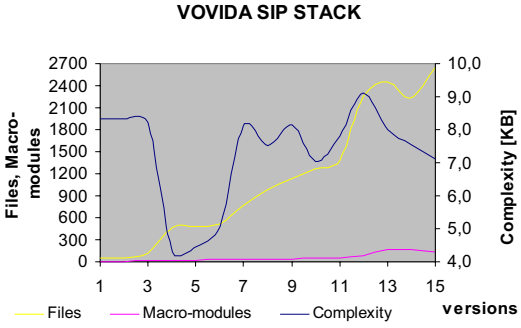

Figure 21 SIP project's observations

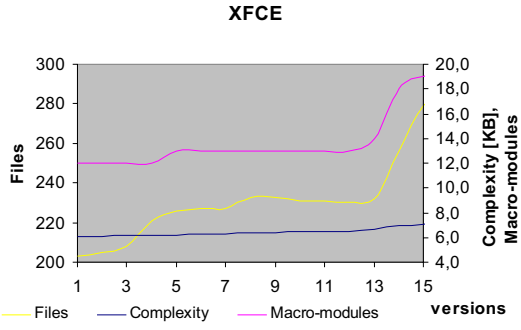

Figure 22 XFCE project's observations