Electronic Communications of the EASST
Volume X (2011)

Proceedings of the Fifth International Workshop on on Foundations and Techniques for Open Source Software Certification (OpenCert 2011)

The Role of Best Practices in Assessing Software Quality

Miguel Regedor, Daniela da Cruz and Pedro Henriques

13 pages

Guest Editors: Luís Soares Barbosa, Dimitrios Settas
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

The Role of Best Practices in Assessing Software Quality

Miguel Regedor[1], Daniela da Cruz[2] and Pedro Henriques[3]

[1] miguelregedor@gmail.com
[2] danieladacruz@di.uminho.pt
[3] prh@di.uminho.pt
Dep. de Informática / CCTC
Universidade do Minho
Braga, Portugal

Abstract: Thousands of open source software (OOS) projects are available for collaboration in platforms like Github or Sourceforge. However, like traditional software, OOS projects have different quality levels. The developer, or the end-user, need to know the quality of a given project before starting the collaboration or its usage—they might of course to trust in the package before taking a decision. In the context of OSS, trustability is a much more sensible concern; mainly end-users usually prefer to pay for proprietary software, to feel more confident in the package quality. OSS projects can be assessed like traditional software packages using the well known software metrics. In this paper we want to go further and propose a finer grain process to do such quality analysis, precisely tuned for this unique development environment. As it is known, along the last years, open source communities have created their own standards and *best practices*. Nevertheless, the classic software metrics do not take into account the *best practices* established by the community. We feel that it could be worthwhile to consider this peculiarity as a complementary source of assessment data. Taking Ruby OSS community and projects as framework, this paper discusses the role of *best practices* in measuring software quality.

Keywords: software metrics, static code analysis, open-source, program comprehension

1 Introduction

Nowadays, Open Source Software (OSS) is well disseminated. Thousands of OSS packages can be found online, and free to download, in Open Source Project Hosting Websites (OSPHW) like SourceForge[1], Google Code[2], or GitHub [3]. Those websites, usually in conjunction with a Version Control System (VCS), make it easy for developers, all around the globe, to collaborate in Open Source Software Projects (OSSP), and also act as a way to make software available to users.

[1] http://sourceforge.net/.
[2] http://code.google.com/.
[3] https://github.com/.

According to NetCraft[4], the market share for top servers across the million busiest sites was 66.82% for the open source web server, Apache, much higher than the 16.87% for Microsoft web servers in May 2010. Even governments started noticing open source, during the last few years, and in some case adopted it[Hah02]. The broad acceptance of OSS means that now OSS is not only used by computer specialists.

John Powell[5] has declared that measuring the savings that people are making in license fees, the open-source industry is worth 60 billion dollars. Matt Asay[6] shares the view that from the customers perspective open source can be now considered the largest software industry in the world. The full review can be found at CNET News[7].

Usually large industries have a strict organization model, that is not the way open source communities operates. Open Source communities work in a kind of *bazaar style*. [RE00] compares the traditional software development process to built cathedrals, few specialized individuals working in isolation. While open source development seemed to resemble a great babbling *bazaar*. But OSS is not developed, all the time, in *bazaar style* and each community can have particular habits. Currently, big open source projects can have companies supporting them. However, most projects are not that big and sometimes it is hard to distinguish the project developers from the project customers/users, because of that bug reports and wanted features can get indistinguishable too. The specification of an open source software project evolves in an organic way [CM07].

Can software that is developed in such chaotic way be trusted as a high quality product? The shock is that in fact the *bazaar style* seemed to work [HS02]. Some big projects, for instance Linux distributions such as Ubuntu[8], are the proof of it. However, how can the quality of this software be measured?

The most basic meaning of software quality is commonly recognized as lack of ”bugs”, and the meeting of the functional requirements. But quality is not simply based on that [GKS+07]. The quality of a software system depends, among other things, on update frequency, quantity of documentation, test coverage, number and type of its dependencies and good programming practices. By analysing those parameters a user can make a better choice when picking software for a specific task [MA07].

When a user/developer finds a new OSSP, for example in GitHub, the things that will most influence the time needed to have a better understanding of the project, to use, or collaborate in it, are the quality of the documentation and the source code readability. Although the OSPHWs provide plenty of useful information about the hosted projects, currently, they do not give a quick answer to the following questions: Does this project have good documentation? Does the code follow standards? How similar is it to other projects?

An OSSP is built up from hundreds, sometimes thousands, of files. It can be coded in many different computer languages. To analyze manually a software project is a very hard and time consuming task, and not all users have the ability to answer the previous questions by looking at the source code [CAH03].

[4] http://news.netcraft.com/archives/2010/05/14/may_2010_web_server_survey.html/, accessed on 2010/12/21.
[5] John Powell is CEO, President, and Co-founder, Alfresco Software Inc.
[6] Matt Asay is chief operating officer at Canonical, the company behind the Ubuntu Linux operating system.
[7] http://news.cnet.com/8301-13505_3-9944923-16.html/ accessed on 2010/12/21.
[8] http://www.ubuntu.com. Ubuntu is a free & open source operating system.

However, open source communities are constantly creating and improving their working methodologies. And even without noticing, communities create rules and best practices. By following those *best practices*, software projects increase their maintainability level.

With that in mind, a system capable of analyzing and measuring a given OSSP, producing detailed quantitative and qualitative reports about it, would enable users to make better choices and, of course, developers to further improve the package.

This paper discusses the concept of Quality when addressing an OSSP, and how to measure it (Section 2) using classic approaches. After that, the notion of *best practices* is introduced and the impact of taking their use into account when assessing an OSSPs is explored. To make this proposal clearer and stronger, Ruby[9] community is taken as a starting target (Section 3). At last but not least, to support our proposal a case-study is shown, in Section 4: seven Ruby OSSP are measured and compared.

2 Assessing Open Source Software

The simplest operation in science and the lowest level of measurement is classification [Kan02].

By assessing OSS we mean to sort OSS projects into an ordinal scale[10] This can be achieved by defining a ranking system[11] and by placing OSS projects into quality categories with respect to certain quality attributes. First we need to find a way of quantifying those OSS quality attributes.

In software, quality is an abstract concept. It is commonly recognized as lack of ”bugs”, and the meeting of the functional requirements. However, quality can be perceived and interpreted differently based on the actual context, objectives and interests of each project. Many software development companies do monitor costumer satisfaction as a quality index, for instance, IBM ranks their software products in levels of CUPRIMDSO [Kan02]:

- Capability/Functionality (refers to the software meeting its functional requirements)
- Usability (refers to the required effort to learn, and operate the software)
- Performance/Efficiency (refers to the software performance and resource consumption)
- Reliability (refers to software fault tolerance and recoverability)
- Instalability/Portability (refers to the required effort to install or transfer the software to another environment)
- Maintainability (refers to the required effort to modify the software)
- Documentation/Information (refers to the coverage and accessibility of the software documentation)
- Service (refers to the company monitoring and service)

[9] Ruby is an open source programming language. Ruby community is relatively young but still very focused on following best practices.

[10] Ordinal scale refers to the measurement operations through which the subjects can be compared in order.

[11] Raking system example: to classify a quality attribute, for instance the project documentation, according to its quality with five, four, three, two or one star.

- Overall (refers to an overall classification based on the other attributes)

Almost every big software company have similar quality attributes. ISO/IEC 9126 provides a framework for the evaluation of software quality (The goal is to achieve quality in use, in other words, quality from the user perspective) [Bev99] IISO/IEC 912 defines six software quality attributes:

- Functionality (refers to the software meeting of the functional requirements)
- Reliability (refers to software fault tolerance and recoverability)
- Usability (refers to the required effort to learn, and operate the software)
- Efficiency (refers to the software performance and resource consumption)
- Maintainability (refers to the required effort to modify the software)
- Portability (refers to the required effort to transfer the software to another environment)

Quality attributes have interrelationships. They can be conflictive[12] or support[13] one another. For example, the higher the functional complexity of the software, the harder it becomes to achieve maintainability [Kan02].

Because of the OSP bazaar style and continuous development process, it is intuitive that the maintainability and documentation attributes have a big influence on the overall quality and continuous progress of an OSP (maintainability and documentation have support relationships with usability, reliability and availability attributes, but might be conflictive with functionality and performance attributes).

Failure to meet functionality often leads to late changes and increased costs in the development process. The software industry and researchers have been mostly interested on testing methodologies that focus on functional requirements and pay little attention to non-functional requirements [CP09].

There are several challenges and difficulties, in assessing non-functional quality attributes for software projects. For example, security is a non-functional requirement that needs to be addressed in every software project. Therefore a badly-written software may be functional, but subject to buffer overflow attacks. Another example is the amount of codebase comments, if the code does not have any comments it will not affect the functional requirements, but it is obvious that it will decrease readability and maintainability [GKS+07].

2.1 Classic Software Metrics

To classify OSS with regards to a certain quality attribute, we need to find which factors influence it. Then we need a way to measure that attribute. If we need to measure we need metrics.

Fortunately, there are around two thousand documented software metrics, but there is few information on how those metrics relate to each other. Most of them simply have different names but give similar information [FN99]. The major challenge is to discover how important

[12] Conflictive, negative influence, if one attribute is high it makes the other one low.
[13] Support, positive influence, if one attribute is high it makes the other one high too.

the information given by those metrics is, if the calculation effort pays off, how to interpret their values and find correlations[14] to assess the quality attributes of an OSP.

2.1.1 Lines of Code

A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements in the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements [CDS86].

2.1.2 Cyclomatic Complexity

The measurement of cyclomatic complexity [McC76] was designed to indicate a program's testability and maintainability. It is the classical graph theory cyclomatic number, indicating the number of regions in a graph. As applied to software, it directly measures the number of linearly independent paths through a program source code.

2.1.3 Fan-In and Fan-Out

Fan-in and fan-out are perhaps the most common design structure metrics, which are based on the ideas of coupling [YC79]:

- *Fan-in* is a count of the modules that call a given module
- *Fan-out* is a count of modules that are called by a given module

In general, modules with a large fan-in are relatively small and simple, and are usually located at the lower layers of the design structure. In contrast, modules that are large and complex are likely to have a small fan-in. There is also the theory that high fan-outs represent a high number of method calls and thus are undesirable, while high fan-ins represent a high level of reuse [WLCR07].

2.1.4 Object-Oriented Metrics

Classes and methods are the basic constructs of OO technology. The amount of function provided by an OO software can be estimated based on the number of identified classes and methods or their variants. Therefore, it is natural that the basic OO metrics are related to classes, methods and their size.

The pertinent question therefore is what should the optimum value be for OO metrics. There may not be one correct answer, but based on his experience in OO software development, Lorenz proposed eleven metrics as OO design metrics called rules of thumb [LK94].

- *Average Method Size (LOC)*: Should be less than 8 LOC for Smalltalk and 24 LOC for C++

[14] Correlation is probably the most widely used statistical method to assess relationships among observational data [Kan02].

- *Average Number of Methods per Class*: Should be less than 20. Bigger averages indicate too much responsibility in too few classes.
- *Average Number of Instance Variables per Class*: Should be less than 6. More instance variables indicate that one class is doing more than it should.
- *Class Hierarchy Nesting Level (Depth of Inheritance Tree, DIT)*: Should be less than 6, starting from the framework classes or the root class.
- *Number of Class/Class Relationships in Each Subsystem*: Should be relatively high. This item relates to high cohesion of classes in the same subsystem. If one or more classes in a subsystem don't interact with many of the other classes, they might be better placed in another subsystem.
- *Average Number of Comment Lines (per Method)*: Should be greater than 1.

3 Best Practices in OSSP development

Open source communities have a tendency to create coding standards. It is a natural and evolutive process. Standards are not rules but instead best practices that are spread through the community and everybody does it that way. Furthermore, best practices discourage:

- Poor performance (due to bad patterns)
- Poor error checking (defensive programming)
- Inconsistent exception handling / Maintainability (long-term quality)

When a developer follows the standards and best practices, the project maintainability is increased. Consequently, project new comers will find it easier to understand the project codebase [Dro02].

However, there is little work done concerned with measuring coding standards by automatic analyzing source code. A plausible explanation for that is the fact that best practices are not a set of immutable rules, they are a continuous evolution and improvement of development methodologies. Communities are constantly creating rules and best practices, even without noticing it. It is not possible to write down a list of best practices without some ambiguities. Nevertheless, it is still possible, to use metrics on the source code and, by analyzing their values, to find hints to help answering weather some methodological approaches were taken into account during the project development process.

At first glance, best practices metrics are for classic metrics as natural as medicine is for science. But, it is not the case. In fact, classic metrics, on their own, do not give much information about a project. In many cases, best practices can be the key to understand what should be the optimum value for a classic metric, for instance, how many lines of code should a ruby method have?

Of course, those questions are subjective. However by analyzing renowned projects, developers opinions and so on, it is possible to find the best practice and that gives a plausible answer to the optimum value.

We believe that, actually, best practices can give a meaning to metrics.

3.1 Best Practices in RoR Projects

Ruby is a dynamic, object oriented, open source programming language created by Yukihiro Matsumoto and public released in 1995. It has an elegant syntax that is natural to read and easy to write. Ruby has drawn devoted coders worldwide. In 2006, Ruby achieved mass acceptance. Moreover, the web framework Ruby on Rails is considered the biggest responsible for Ruby popularity (tens of thousands of Rails applications are online).

Ruby and Ruby on Rails community members are, in general, addicted to best practices. However, in reality, many of those best practices are studied development methodologies. For instance, the majority of Ruby on Rails book authors speak about automated tests, written using specific DSLs, like Cucumber or Rspec. It is also common to associate Ruby on Rails with Behaviour Driven Development (BDD) and Agile methodologies.

Because of all this, the ruby community has great potential to be a starting point to understand the role of best practices, its benefits and how to measure it. In fact, there is already some work done.

The web site Rails Best Practices[15], works in similar way to a web forum and its objective is to engage developers to discuss which practices should be considered best practices to follow, when building a RoR web application. The community involved with this web site is committed to build a gem[16] that produces a report about a given project.

3.2 Ruby Best Practices Examples

But what is is a best practice after all? Best practices can be related to code formatting:

- *Use two spaces to indent code and no tabs*, it is a matter of taste but every worthy ruby developer do it that way.
- *Remove trailing whitespace*, trailing whitespace makes noises in version control systems.

Can be related to syntax:

- *Avoid return where not required*.
- *Suppress superfluous parentheses*, when calling methods, but keep them when calling functions(when you use the return value in the same line).

Can be related to naming:

- *Use snake_case for methods*.

[15] http://www.rails-bestpractices.com/ is a web site created by Richard Huang, it was inspired by Wen-Tien Chang talk given at Kungfu RailsConf 2009 in Shanghai. Slides can be found here http://www.slideshare.net/ihower/rails-best-practices.

[16] Ruby Libraries are called gems. Ruby gems can be easily managed using rubygems (rubygems is for Ruby as aptitude is for Debian or cpan for perl).

- *Other method naming conventions*: Use map over collect, find over detect, find_all over select, size over length.

And can also be specific to a framework, rails best practices:

- *Law of Demeter*, A model should only talk to its immediate association.
- *Move code into controller*, according to MVC architecture, there should not be logic codes in view.
- *Isolate seed data*, do not insert seed data during migrations, a rake task[17] can be used instead.
- *Do not use default route*, When using a RESTful design. The default RoR routes can cause a security problems.
- *Replace Complex Creation with Factory Method*, Sometimes you will build a complex model with params, current_user and other logics in controller, but it makes your controller too big, you should move them into model with a factory method.

4 Assessing Ruby on Rails Projects

After deciding that some *procedure* is a *best practice*, it would be handy to find a way to automatically verify weather that practice is being followed by the developers of a given project. With that in mind, an open source ruby gem was created (by the authors of Rails best practices web site) with the objective of automatically produce a report that shows where, in the source code, a project is failing to obey to consensual practices. At moment of writing, this gem can check for 28 kinds of best practices (from the 70 described in that web site).

However, one of the first things that we have noticed when we have applied this gem to OSS projects, is that the biggest and renowned projects have much more errors than the small and unknown projects. This nonsense has a simple interpretation. Small projects (like the majority of RoR projects found in github) are simple software packages, often developed by a single user. These applications are so simple that many times the code is almost entirely created by RoR code generators. Usually, when code is not written by humans, it has few mistakes concerning those recommendations.

4.1 First Study

Having the above into account, we decided to run the rail best practices gem on similar RoR (Ruby on Rails) projects. Seven *time tracking* or *project management* open source systems were chosen. After running the gem and counting not best practices (NBPs)[18] occurrences, the following results were obtained:

[17] Rakefiles work in similar way to Makefiles but are written in ruby. It is a simple way to write code to automate repetitive tasks.

[18] In fact, Rails best practices gem do not find best practices in the source code. It does the opposite, it discovers when the code is not written according to a best practice, in other words, it identifies bad practices (similar to the detection of code smells). We decided to name those occurrences NBP.

Rails Best Practices Results							
Best Practice	**A**	**B**	**C**	**D**	**F**	**G**	**H**
Add model virtual attribute	-	2	7	-	-	5	4
Always add db index	-	-	-	43	-	-	51
Isolate seed data	-	-	-	-	-	79	17
Law of demeter	20	38	45	6	30	164	85
Move code into controller	-	-	-	-	2	-	4
Move code into model	-	26	-	7	1	3	19
Move model logic into model	-	-	76	11	11	98	100
Move finder to named_scope	-	4	9	2	4	25	-
Needless deep nesting	-	-	-	1	-	-	-
Not use default root	-	1	1	-	1	1	1
Notes use query attribute	-	2	-	-	-	-	-
Overuse route customizations	-	-	2	4	-	2	2
Remove trailing whitespace	68	57	126	110	330	316	100
Use factory method	-	15	9	5	1	8	19
Replace instance var with local var	13	-	70	239	142	31	100
Use before_filter	-	7	9	8	8	19	23
Wrong email content_type	-	3	-	-	-	-	-
Use query attribute	-	-	11	5	8	29	6
Use say with time in migrations	-	-	24	-	10	23	56
Use scopes access	-	-	-	-	-	-	04
User model association	-	-	12	9	-	1	21
Keep finders on their own model	8	4	1	-	11	-	-
Total	109	156	402	450	559	834	864

A: Rubytime , *B:* Notes , *C:* Tracks , *D:* Handy Ant , *F:* Retrospectiva , *G:* Redmine , *H:* Clockingit
Figures shown represent the number of times a project do not follow a best practice; is expected that *smaller the number, better the project*.

Table 1: Results obtained by running the *best practices analyzer gem* on the 7 Open Source Projects chosen (data produced on April, 2011).

Rubytime seems to have the best results and Clockingit the worst. The fact is that very good user reviews can be found about Rubytime. However, Tracks obtained an unexpected high score, since it has been very low maintained (old code has higher probability of not following the current best practices). As explained before, those values are not really measuring if a project follows best practices but instead measuring when it fails. This should be also be taken into consideration.

The most evident problem here is that best practices are not being weighted and neither the size of the project considered. For instance, if the developers have the habit of leaving trailing white spaces, the occurrences of this will obviously be related to the size of the project. On the other hand, it is a best practice to remove the default route generated by rails, independently of the project size this is true or false, there is no way to leave the route two times. So, if developers do not take into account those two best practices, when the project grows, the number of trailing spaces will increase and the results will show more NBPs, but the other one will always be only one NBP. Because of that we can get twisted results.

To avoid this, the projects were sized. The size attribute is based on the quantity of models and controllers in the project. After that, we divided the values previously obtained by the project size. By doing that, a new set of results emerge.

Rails Best Practices Results							
Best Practice	**A**	**B**	**C**	**D**	**F**	**G**	**H**
Total	109	156	402	450	559	834	864
Total Without Trailing Whitespace	41	99	276	340	229	518	764
Project Size	12	11	11	29	26	58	31
Total / Project Size	9	14	37	16	23	15	28
Total Without Trailing Whitespace / Project Size	3	9	25	12	9	9	25

A: Rubytime. , *B:* Notes , *C:* Tracks , *D:* Handy Ant , *F:* Retrospectiva , *G:* Redmine , *H:* Clockingit

Table 2: Results obtained by running the *best practices analyzer gem* on the 7 Open Source Projects chosen, after normalization (data produced on April, 2011).

Those results are much more likely to be helpful in terms of understanding if a project is or is not following best practices. The numbers reflect much more the community reviews and the our expectations.

4.2 Second Study

After the first study reported above, we felt that it was time to make a bigger one; we should repeat the experiment over a larger sample. In addition, there was the need to define an objective quality metric to compare the metrics results with.

For the second study, we selected 40 Ruby on Rails projects hosted in github and decided to consider the number of followers and forks, that each project has on github, as a *project reputation* metric.

The objective was to prove that a negative correlation exists, between the NBPs of a project and its followers and forks. The previous study has shown us the need to apply different weights to each NBP. By diving the NBPs by the project size, in the first study, seemed like we got better results. However, not all NBPs depend on the project size. Therefore, we altered the rails best practices gem to make it possible to know how much files of the project were analyzed by each rails best practice checker.

Basically, after collecting the GitHub URLs for each project, we followed the next steps:

- *Retrieve GitHub information*, in this step we get the followers and forks(and more info that might me used in further analyses).
- *Download the project repository*.
- *Run rails best practices gems*, at this point, we get the non witdh NBPs and files checked for each one of the 28 checkers.
- *Calculate the Weighted Global NBPs*, we divide each checker NBPs by the files checked and sum it.

Next, an excerpt of the obtained table is shown:

Rails Best Practices Results											
Projects	**Forks**	**Watchers**	C1	C1 F.	**W. C1**	C2	C2 F.	**W. C1**	...	T. NBPs	**W. T. NBPs**
Rails Admin	30	2478	0	141	**0**	0	37	**0**	...	50	**739**
Rubytime	12	82	24	161	**149**	0	134	**0**	...	146	**1334**
Redmine	30	1781	49	996	**49**	1	362	**2**	...	884	**1402**
BrowserCMS	30	784	11	234	**47**	0	216	**0**	...	268	**1510**
Tracks	17	87	46	842	**54**	15	271	**55**	...	569	**2810**
...	...	...	...	...	...	...	...	...	...	...	...

C(x): The rails best practices gem has 29 checkers(when this study was carried), each one tries to find occurrences of a different nbp in the project.
C(x) Files: The number of files in the project, where it tried to find nbps (for instance, some checkers may only be concerned with html files, some other checker nbps my only occur in model files, etc)
W. C(x): Weighted C(x) = C(X) / C(x)Files * 1000 (A really small number is added to each variable to avoid divisions by zero).

Table 3: Results obtained by running the *best practices analyzer gem* on the 40 Open Source Projects chosen, from GitHub (data produced on April, 2011). The full table can be found at www.bestpracticesstudy.gorgeouscode.com

4.3 Results

After building this table, containing the results for the 40 projects we could easily try to find correlations between columns. We discovered that the average correlation index, for the weighted C(x) columns, is -0.2. Only three of the weighted C(x) columns do not have negative correlation. This is quite good, considering the fact that there an explanation for it. Those three checkers were trying to find nbps that almost non of the projects were committing, so there is no correlation.

The most important results are in the next table:

Correlations		
	Total NBPs	**Total Weighted NBPs**
Forks	0.14	-0.53
Watchers	0.07	-0.40

Table 4: The full table can be found at www.bestpracticesstudy.gorgeouscode.com

This correlations indexes show that if we just count the nbps there is no relation between them and the number of forks and watchers. Nevertheless, the Weighted NBPs have a quite perceptible negative correlation both with watchers and forks.

To notice, that the forks correlation is bigger. We believe that it happens is because forking a project shows intensions of digging into the code and, of course, it easier to understand others code when it follows good practices.

For future work, we are considering more correlations with other variables that are already available, but we haven't used yet. The most relevant ones: the number of commiters, starting date of the project, last commit data, and total number of commits. We believe that those variable can strongly be related withe the forks, watchers and of course, in the end, the quality of the project.

5 Conclusion

at this point is easy to convert the Global NBPs into a 5 stars score.

Nowadays, thousands of open-source software packages can be found and freely downloaded online. github is a web-based hosting service for projects that use the Git revision control system. It hosts more than 1 million open-source projects.

There is little work done concerning the measurement of coding standards by automatic analyzing source code. We strongly believe that some research and development should be done in this direction. Along the paper we gave arguments in order to make evident that it is worthwhile to detect on the source code that the author follows the best practices recommended by the respective community.

In this particular context, Ruby Community, there is already some work done. The reports generated by the existent source code analyzers, can spot the occurrences of bad smells but this is not enough. There is the need to interpret those results to end up with a high level quality statement. By comparing some projects, it was possible to start understanding how to interpret those values. For instance, the *size of the project* should be taken into consideration (it is intuitive that a project with 10 lines of code and 10 errors is worse than a project with 1000 and 20 errors). From the study we carried out and, described in the paper, we also have learned that each best practice has a different importance level — 1 error that affects security or performance is, for sure, worse than 10 errors related to indentation; or 10 errors related to naming conventions are worse than the indentation mistakes).

We do believe that by analyzing a massive amount of open source projects, it is possible to create a new set of metrics capable of quantify the standards followed by a given project, judge the impact of the metrics evaluated and consequently assess its level of maintainability.

The future work is to develop a system capable of automatically produce quality reports about a given OSSP combining traditional SW metrics with best practices analysis.

This system will enable users to make better choices about what software to use and help developers to improve their software.

Bibliography

[Bev99] N. Bevan. Quality in use: meeting user needs for quality. *Journal of Systems and Software* 49(1):89–96, 1999.

[CAH03] *Defining open source software project success*. 2003.

[CDS86] S. Conte, H. Dunsmore, Y. Shen. *Software engineering metrics and models*. 1986.

[CM07] A. Capiluppi, M. Michlmayr. From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects. *INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING -PUBLICATIONS- IFIP* 234/2007:31–44, 2007.

[CP09] L. Chung, J. do Prado Leite. On non-functional requirements in software engineering. *Conceptual Modeling: Foundations and Applications*, pp. 363–379, 2009.

[Dro02] R. Dromey. A model for software product quality. *Software Engineering, IEEE Transactions on* 21(2):146–162, 2002.

[FN99] N. Fenton, M. Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software* 47(2-3):149–157, 1999.

[GKS+07] *Software quality assessment of open source software*. Athens University of Economics and Business, Patission 76, Athens, Greece, 2007.

[Hah02] R. Hahn. *Government policy toward open source software*. Brookings Institution Press, Washington, DC, USA, 2002.

[HS02] *High quality and open source software practices*. 2002.

[Kan02] S. Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[LK94] M. Lorenz, J. Kidd. *Object-oriented software metrics: a practical guide*. 1994.

[MA07] A. Marchenko, P. Abrahamsson. Predicting software defect density: a case study on automated static code analysis. *Agile Processes in Software Engineering and Extreme Programming* 4536/2007:137–140, 2007.

[McC76] T. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pp. 308–320, 1976.

[RE00] E. Raymond, T. Enterprises. The Cathedral and the Bazaar. *KNOWLEDGE, TECHNOLOGY AND POLICY* 12:1–35, 2000.

[WLCR07] Y. Wang, Q. Li, P. Chen, C. Ren. Dynamic fan-in and fan-out metrics for program comprehension. *Journal of Shanghai University (English Edition)* 11(5):474–479, 2007.

[YC79] E. Yourdon, L. Constantine. Structured design. Fundamentals of a discipline of computer program and systems design. 1979.