The Role of Best Practices in Assessing Software Quality

Miguel Regedor
Daniela da Cruz
Pedro Henriques

Minho University, miguelregedor@gmail.com

Abstract. Thousands of open source software projects are available for collaboration in platforms like Github or Sourceforge. Along the years, open source communities created their own standards and best practices. However, like traditional software, open source software projects have different quality levels. The traditional way to assess the quality of a software project is based on software metrics. Nevertheless, the classic software metrics do not take into account the *best practices* established by the community. This paper discusses what should be taken into account to measure a project in terms of best practices, having as case study the open source ruby language.

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.

Nevertheless, OSPHW are not the only prof of OSS establishment. 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[9]. The big 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]

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

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*. [15] 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*. Currently, big open source projects can have companies supporting them, and each community can have particular habits. 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 [2].

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 [10]. 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 [8]. 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 [13].

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 OSPH websites 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? A OSSP is built up from hundreds, sometimes thousands, of files. It can be coded in many different computer languages. To manually analyse 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 [5]. 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 automatically analysing and producing reports about a given OSSP would enable developers to further improve their software, and users to make better choices.

[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.

This paper discusses the software quality definition and how to measure it (Section 2).

After that, it explores the idea of measuring community best practices for the development of OSSPs, having the ruby[9] community as a starting target (Section 3).

At last but not least, a study case 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 [11].

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 [11]:

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

Almost every sofware 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) [1] IISO/IEC 912 defines six software quality attributes:

[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.

- 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 [11].

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 [3].

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 [8].

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 [7]. The major challenge is to discover how important 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.

[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.
[14] Correlation is probably the most widely used statistical method to assess relationships among observational data [11].

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 [4].

Cyclomatic Complexity The measurement of cyclomatic complexity [14] 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.

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 [17]:

- *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 [16].

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 [12].

- *Average Method Size (LOC)*: Should be less than 8 LOC for Smalltalk and 24 LOC for C++
- *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 Subsystem/Subsystem Relationships*: Should be less than the number in metric 6.
- *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.

2.2 Best Practices

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 follow the standards and best practices, the project maintainability is increased. Consequently, project new comers will find it easier to understand the project codebase [6]. However, there is few work done related to measuring coding standards by automatic analysing 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 analysing their values, to find hints to help answering if a set of methodologies were taken into account during the project development process.

At first glance, best practices metrics are for classic metrics as natural medicine is for science. But, it is not the case. In fact, classic metrics, on theirs 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, these questions is subjective. However by analysing 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.

In the end, best practices can give a meaning to metrics.

3 Best Practices in Ruby on Rails 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.

3.1 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.*
- *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 can be used instead.

[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).

- *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 something is a best practice, it would be handy to find a way to automatically verify if that practice is being followed by a given project. Having that in mind, a open source ruby gem was created (by the rails best practices web site authors) with the objective of automatically produce a report that shows where, in the source code, is a project failing to consensual best practices. At moment of writing, this gem can check for 28 kinds of best practices (from about 70 described on the web site).

However, one of the first things to notice when using this gem, is that the biggest and renowned projects have more errors than the unknown projects. This nonsense has a simple explication. The majority of RoR projects found in github are simple projects, in most cases, developed by a single user. These applications are so simple that sometimes the code is almost entirely created by RoR code generators. Usually, when code is not written by humans it has few mistakes.

Having the above into account, we decided to run the rail best practices gem on similar Ruby on Rails projects. Seven time tracking and/or project management open source system were chosen. After running the gem and counting the occurrences, the following results were obtained.

Rails Best Practices Results							
Best Practice	**Rubytime**	**Notes**	**Tracks**	**Handy Ant**	**Retrospectiva**	**Redmine**	**Clockingit**
Add model virtual attribute[a]	-	2	7	-	-	5	4
Always add db index[b]	-	-	-	43	-	-	51
Isolate seed data[c]	-	-	-	-	-	79	17
Law of demeter[d]	20	38	45	6	30	164	85
Move code into controller[e]	-	-	-	-	2	-	4
Move code into model[f]	-	26	-	7	1	3	19
Move model logic into model[g]	-	-	76	11	11	98	100
Move finder to named_scope[h]	-	4	9	2	4	25	-
Needless deep nesting[i]	-	-	-	1	-	-	-
Not use default root[j]	-	1	1	-	1	1	1
Notes use query attribute[l]	-	2	-	-	-	-	-
Overuse route customizations[m]	-	-	2	4	-	2	2
Remove trailing whitespace[n]	68	57	126	110	330	316	100
Use factory method[o]	-	15	9	5	1	8	19
Replace instance var with local var[p]	13	-	70	239	142	31	100
Use before_filter[q]	-	7	9	8	8	19	23
Wrong email content_type[r]	-	3	-	-	-	-	-
Use query attribute[s]	-	-	11	5	8	29	6
Use say with time in migrations[t]	-	-	24	-	10	23	56
Use scopes access[u]	-	-	-	-	-	-	04
User model association[v]	-	-	12	9	-	1	21
Keep finders on their own model[x]	8	4	1	-	11	-	-
Total[z]	109	156	402	450	559	834	864

[a] Add model virtual attribute
[b] Always add db index
[c] Isolate seed data
[d] Law of demeter
[e] Move code into controller
[f] Move code into model
[g] Move model logic into model
[h] Move finder to named_scope
[i] Needless deep nesting
[j] Not use default root
[l] Notes use query attribute
[m] Overuse route customizations
[n] Remove trailing whitespace
[o] Use factory method
[p] Replace instance var with local var
[q] Use before_filter
[r] Wrong email content_type
[s] Use query attribute
[t] Use say with time in migrations
[u] Use scopes access
[v] User model association
[x] Keep finders on their own model
[z] Total

Table 1. Results obtained by running the best practices analizer gem on 7 Open Source Project/Time Management Applications, this data was produced on April, 2011.

Rubytime seems to have the best results and Clockingit the worst. The fact is that very good reviews can be found about Rubytime. However, Tracks obtained an unexpected high rake, since it has been very low maintained and it is getting deprecated. The biggest problem here is that best practices are not being weighted and neither the size of the project. For instance, if the developers have the habit of leaving trailing white spaces, the occurrences of this will obvious 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. 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 previous obtained value by the project size. By doing that, a new set of results emerge.

Rails Best Practices Results							
Best Practice	**Rubytime**	**Notes**	**Tracks**	**Handy Ant**	**Retrospectiva**	**Redmine**	**Clockingit**
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

Table 2. Results obtained by running the best practices analizer gem on 7 Open Source Project/Time Management Applications, this data was 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.

However, those results are not really measuring if a project follows best practices but instead measuring when it fails to it. This should be also be taken into consideration.

5 Conclusion

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.

When a user/developer finds a new OSSP in github the things that will most influence the time needed to understand the project, to use or collaborate with it are the quality of the documentation and the codebase readability. Although OSPH websites provide a lot of useful information about the hosted projects they do not currently give a quick answer to the quality of it Is it easy to understand this project source code? Does it have good documentation? Does the code follow standards? Does it have high quality? A OSSP is built up from hundreds, sometimes thousands, of files. It can be coded in many different computer languages. To manually analyse 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 codebase.

There is few work done in what relates to measuring coding standards by automatic analysing source code. Nevertheless, in the ruby community, there is already some work done. The generated reports by the existent source code

analizers, can spot the occurrences of bad smells but that is not enough. There is the need to interpret those results to end up with a high level classification.

By comparing some projects it was possible to realize 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 worst than a project with 1000 and 20 errors), also that each best practice have a different importance level (1 error that can affect security or performance maybe worst than 10 errors related to indentation, and 10 errors related to naming conventions can be worst than the last one).

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

The future work is to develop a system capable of automatically analysing and producing reports about a given OSSP.

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

References

1. Bevan, N.: Quality in use: meeting user needs for quality. Journal of Systems and Software 49(1), 89–96 (1999)
2. Capiluppi, A., Michlmayr, M.: From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects. ? ?, 31–44 (?)
3. Chung, L., do Prado Leite, J.: On non-functional requirements in software engineering. Conceptual Modeling: Foundations and Applications pp. 363–379 (2009)
4. Conte, S., Dunsmore, H., Shen, Y.: Software engineering metrics and models (1986)
5. Crowston, K., Annabi, H., Howison, J.: Defining open source software project success. In: Proceedings of the 24th international conference on information systems (icis 2003). pp. 327–340. Citeseer (2003)
6. Dromey, R.: A model for software product quality. Software Engineering, IEEE Transactions on 21(2), 146–162 (2002)
7. Fenton, N., Neil, M.: Software metrics: successes, failures and new directions. Journal of Systems and Software 47(2-3), 149–157 (1999)
8. Gousios, G., Karakoidas, V., Stroggylos, K., Louridas, P., Vlachos, V., Spinellis, D.: Software quality assesment of open source software. In: Proceedings of the 11th Panhellenic Conference on Informatics. Citeseer, ?, ? (2007)
9. Hahn, R.: Government policy toward open source software. Brookings Institution Press, ? (2002)
10. Halloran, T., Scherlis, W.: High quality and open source software practices. In: Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering (2002)
11. Kan, S.: Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, ? (2002)
12. Lorenz, M., Kidd, J.: Object-oriented software metrics: a practical guide (1994)
13. Marchenko, A., Abrahamsson, P.: Predicting software defect density: a case study on automated static code analysis. Agile Processes in Software Engineering and Extreme Programming ?, 137–140 (2007)

14. McCabe, T.: A complexity measure. IEEE Transactions on software Engineering pp. 308–320 (1976)
15. Raymond, E., Enterprises, T.: The Cathedral and the Bazaar. ? ?, 1–35 (2000)
16. Wang, Y., Li, Q., Chen, P., Ren, C.: Dynamic fan-in and fan-out metrics for program comprehension. Journal of Shanghai University (English Edition) 11(5), 474–479 (2007)
17. Yourdon, E., Constantine, L.: Structured design. Fundamentals of a discipline of computer program and systems design (1979)