Analysing and Measuring Open Source Projects, state-of-the-art

MIGUEL REGEDOR
Universidade do Minho

Thousands of open source software projects are available for collaboration in platforms like Github or Sourceforge. However, there is no systematic information about the quality of those projects.

Few users have the ability to assess the quality of a project by looking at source code. An application able to perform automatic analysis of a software package and to generate a hight level overview of the code would enable users to make better choices about what software to use and help developers by giving hints on how to improve their software.

This paper discusses what should be taken into account when developing such application and what benefits can be expected.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.8 [**Software Engineering**]: Metrics; F.3.0 [**Logics and meanings of programs**]: General; K.7.3 [**The Computing Profession**]: Testing, Certification, and Licensing

General Terms: Measurement
Additional Key Words and Phrases: Static code analysis, Software Metrics, Open-Source, MI-STAR, Universidade do Minho

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[Hahn 2002]. 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

[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.

Author full name: Carlos Miguel Janeiro Regedor Dias da Torre
Author email: miguelregedor@gmail.com
Universidade do Minho - Master Course on Informatics - State of the Art Reports 2011

licence 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*. [Raymond and Enterprises 2000] 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 [Capiluppi and Michlmayr].

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 [Halloran and Scherlis 2002]. Some big projects, for instance Linux distributions such as Ubuntu[8], are the proof of it. However, how can the quality of this sofware 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 [Gousios et al. 2007]. 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 [Marchenko and Abrahamsson 2007].

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 [Crowston et al. 2003].

With that in mind, a system capable of automatically analysing and producing reports about a given OSSP would enable users to make better choices, and developers to further improve their software. Furthermore, I am, currently, writing my master thesis and engage myself to create an application capable of automatic anal-

[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.

yse and measure GitHub public projects. This paper compares the various OSPHW and gives to know the main reasons why I chose Github for my future work. After that, it discusses software quality definition and how to measure it using software metrics. At last but not least, some already existing tools are presented.

2. OPEN SOURCE PROJECT HOSTING PLATFORMS

Open Source Project Hosting Web Sites					
Name	Established	Available VCS	Users	Projects	Alexa rank [a]
SourceForge	1999	CVS SVN Bazar GIT Mercurial	2,000,000	236,319	136
GitHub	2008	GIT	505,000	1,516,000	742
Google Code	2006	SVN Mercurial	?	250,000	900[b]
Code Plex	2006	SVN Microsoft TFS Mercurial	151,782	15.955	2,343
Assembla	2006	SVN GIT	180,000	60,000	6,628
Launchpad	2005	Bazar	1,140,345	19,016	12,466
BerliOS	2000	CVS SVN GIT Mercurial	47,285	5,448	17,299
Bitbucket	2008	Mercurial	51,600	27,769	12,047
Gitorious	2008	GIT	?	8,336	28,531
GNU Savannah	2000	CVS SVN Bazar Arch GIT Mercurial	48,593	3,233	48,286

Data retrieved on 2010-12-20, from each of the OSPH Websites, and using Alexa rank website.
[a] Alexa rank represents the approximate number of websites, in the world, that have a higher popularity than the given site (the smaller the better).
[b] This value is an approximation.

Table I. Open Source Project Hosting Websites

An open source project hosting platform is the central tool that supports and coordinates the development of an open source project, normally it is in a form of a website.

Since 1999 (year that SourceForge was launched), many OSPHW were created to host open source projects. OSPH sites offer different features, like codebase [9]

[9]The term codebase means the whole collection of source code used to build a particular application or component.

hosting (a project codebase is typically stored in a source control repository), code review, bug tracking, web hosting, wiki, mailing list, etc [Binkley et al. 2006].

Table I shows a list of the most well known OSPHW. By looking at this table we can see that SourceForge is the most well established OSPHW. It is also one of oldest, hosts more than 230,000 projects and has more than 2 million registered users [Christley and Madey 2005].

GitHub is one of the youngest OSPHW (launched in 2008). However, in only two years, it drew more than 500,000 users (one quarter of sourceforge users) and is hosting more than 1,500,000 projects. The only version control system provided by GitHub is git[10]. Because GitHub projects are in fact git repositories, it is incredibly easy to make branches and merges in GitHub. Although branching was considered a bad practice, it turned out that using git, it can in fact improve the developers collaboration and organization. Hosting a Git repository is not hard but co-ordinating efforts of forking and merging amongst people is tough. With a system like Github, it becomes a lot easier [Cooper]. However, the main reason for GitHub popularity are the social aspects. Users and projects have public profiles and activity feeds which display activity on public projects such as commits, comments, forks, etc. Furthermore, with so many high profile projects on board (jQuery, reddit, Sparkle, curl, Ruby on Rails, node.js, ClickToFlash, Erlang/OTP, CakePHP, Redis), it is easy to imagine that GitHub could be the next SourceForge.

The above are the reasons why I believe that to build a software, like the one described in the introduction, with focus on GitHub projects, would benefit a big open-source community.

3. ASSESSING OPEN SOURCE SOFTWARE

The simplest operation in science and the lowest level of measurement is classification [Kan 2002].

By assessing OSS we mean to sort OSS projects into an ordinal scale[11] This can be achieved by defining a ranking system[12] 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.

3.1 Software quality

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 [Kan 2002]:

—Capability/Functionality (refers to the software meeting its functional require-

[10]`http://git-scm.com/`. Git is a free & open source, distributed version control system that Linus Torvalds developed to help manage Linux kernel development.
[11]Ordinal scale refers to the measurement operations through which the subjects can be compared in order.
[12]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.

ments)

—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) [Bevan 1999] 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[13] or support[14] one another. For example, the higher the functional complexity of the software, the harder it becomes to achieve maintainability [Kan 2002].

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 [Chung and do Prado Leite 2009].

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

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

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 [Gousios et al. 2007].

A systematic way to measure non-functional attributes on software projects would benefit software development in general.

3.2 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 [Fenton and Neil 1999]. The major challenge, to create a systematic way to measure the quality (based on non-functional attributes) of sofware project, 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[15] to assess the quality attributes of an OSP.

3.2.1 *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. [Conte et al. 1986] The lines of code (LOC) metric is anything but simple. The major problem comes from the ambiguity of the operational definition, the actual counting. In the early days of Assembler programming, in which one physical line was the same as one instruction, the LOC definition was clear. With the availability of high-level languages the one-to-one correspondence broke down. Differences between physical lines and instruction statements (or logical lines of code) and differences among languages contribute to the huge variations in counting LOC. For instance, Boehm [Boehm 2009] counts lines as physical lines and includes executable lines, data definitions, and comments. Even within the same language, the methods and algorithms used by different counting tools can cause significant differences in the final counts.

Next, the most common variations are described [Jones and Jones 1986]:

—Count only executable lines.

—Count executable lines plus data definitions.

—Count executable lines, data definitions, and comments.

—Count executable lines, data definitions, comments, and job control language.

—Count lines as physical lines on an input screen.

—Count lines as terminated by logical delimiters.

[15]Correlation is probably the most widely used statistical method to assess relationships among observational data [Kan 2002].

3.2.2 *Cyclomatic Complexity.* The measurement of cyclomatic complexity [McCabe 1976] 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. Basically it counts how many different executions a program can have. For instance, one "if" statement will double the number of paths. Therefore, a high number of control statements (ifs, loops, etc) in a program source code will result in a high cyclomatic complexity. As such it can be used to indicate the effort required to test a program. To determine the paths, the program procedure is represented as a strongly connected graph with unique entry and exit points. The general formula to compute cyclomatic complexity is:

$$M = V(g) = e - n + 2p$$

where

—$V(g)$ is the cyclomatic number of g
—e is the number of edges
—n is the number of nodes
—p is the number of unconnected parts of the graph

3.2.3 *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 [Yourdon and Constantine 1979]:

—*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.[Wang et al. 2007].

3.2.4 *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 [Lorenz and Kidd 1994].

—*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.

3.2.5 *Coding Standards.* Open source communities have a tendency to create create coding standards. It is a natural process. Standards are not rules but instead good practices that are spread through the community and everybody does it that way. In addition, coding standards can discourage the following problems:

—Poor performance (due to bad patterns)

—Poor error checking (defensive programming)

—Inconsistent exception handling / Maintainability (long-term quality)

Projects following standards increase their maintainability, and make it easier for project new comers [Dromey 2002].

There is almost no work done in what relates to measuring coding standards by automatic analysing source code. I do believe that by analysing a massive amount of open source software projects, for instance by analysing GitGub projects, patterns can be found. In addition, it can lead into the creation of new set of metrics capable of quantify the standards followed by a given project.

3.3 Available Tools

Tools are intended to make a task easier. Codebase analysis tools are designed turn the task of analyse and measure source code easier, Those tools help in the process of finding software flaws and can also serve as aids for assessing the quality of software.

Next a list of some free tools:

—FindBugs find "bugs" in Java Programs

—FxCop (Microsoft) analyses managed code assemblies and reports information about the assemblies

—PMD scans Java source code and looks for potential code problems

—PreFast (Microsoft) is a static analysis tool that identifies defects in C/C++ programs

—RATS (Fortify) scans C, C++, Perl, PHP and Python source code for security problems like buffer overflows and Time Of Check, Time Of Use race conditions

—SWAAT simplistic tool for Java, JSP, ASP .Net, and PHP

—Flawfinder scans C and C++

Tools like the ones listed above analyze code without executing it, but point out what they consider to be potential weaknesses. The most typical example of what those tools can find probably is calls to the emphgets function in the C programming language: this function is inherently insecure and can lead to buffer overflows. Specially crafted user input values can, for instance, allow an attacker to access or modify confidential data or even take control of any computer executing that piece of software.

Because these tools need to ”understand” the code being analyzed, they are necessarily very language specific. Furthermore, when analyzing a large amount of software projects, for instance Github projects, it is of great importance to have a previous knowledge of what kind of projects are going to be found (programing languages, frameworks, and other attributes), this way tools can be adapted to get better results.

4. 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 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 codebase.

My future work is to develop a system capable of automatically analysing and producing reports about a given OSSP. As this survey shown tools like this are usually language specific and so this system will have to be GitHub specific. Since the predominant languages in GitHub projects are OO, I should pay particular attention to OO related metrics and measuring tools. There is almost no work done in what relates to measuring coding standards by automatic analysing source code. However, I 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.

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

REFERENCES

Asay, M. 2008. The open-source industry is worth $60 billion - CNET news. http://news.cnet.com/8301-13505_3-9944923-16.html.

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

Binkley, D. 2007. Source code analysis: A road map. In *FOSE '07: 2007 Future of Software Engineering.* IEEE Computer Society, Washington, DC, USA, 104–119.

Binkley, D., Harman, M., and Krinke, J. 2006. Animated visualisation of static analysis: Characterising, explaining and exploiting the approximate nature of static analysis. In *6th*

International Workshop on Source Code Analysis and Manipulation (SCAM 06). Philadelphia, Pennsylvania, USA. Citeseer, 43–52.

Boehm, B. 2009. *Software engineering economics.* Number 1. IEEE.

Capiluppi, A. and Michlmayr, M. ? From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects. *? ?*, 31–44.

Christley, S. and Madey, G. 2005. Collection of activity data for sourceforge projects. *Notre Dame, IN, Dept. of Computer Science and Engineering, University of Notre Dame ?*, ?

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

Code, G. 2010. Google Code Web Site. http://code.google.com/.

CodePlex. 2010. CodePlex Web Site. http://www.codeplex.com/.

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

Cooper, P. GitHub officially launches: Git hosting A-Go-Go! http://www.rubyinside.com/github-officially-launches-git-hosting-a-go-go-853.html.

Crowston, K., Annabi, H., and Howison, J. 2003. Defining open source software project success. In *Proceedings of the 24th international conference on information systems (icis 2003).* Citeseer, 327–340.

Curtis, B. 1981. Chapter 12 – the measurement of software quality and complexity. In *Software Metrics: An Analysis and Evaluation*, A. J. Perlis, F. Sayward, and M. Shaw, Eds. MIT-Press.

Dromey, R. 2002. A model for software product quality. *Software Engineering, IEEE Transactions on 21,* 2, 146–162.

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

GitHub. 2010. GitHub Web Site. https://github.com/.

Gousios, G., Karakoidas, V., Stroggylos, K., Louridas, P., Vlachos, V., and Spinellis, D. 2007. Software quality assesment of open source software. In *Proceedings of the 11th Panhellenic Conference on Informatics.* Citeseer, ?, ?

Hahn, R. 2002. *Government policy toward open source software.* Brookings Institution Press, ?

Halloran, T. and Scherlis, W. 2002. High quality and open source software practices. In *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering.*

Hofer, T. 2010. Evaluating Static Source Code Analysis Tools. Ph.D. thesis, École Polytechnique Fédérale de Lausanne.

Hunt, F. and Johnson, P. 2002. On the Pareto distribution of Sourceforge projects. In *Proceedings of the Open Source Software Development Workshop.* 122–129.

Jones, C. and Jones, C. 1986. *Programming productivity.* McGraw-Hill New York.

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

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

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

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

Perlis, A. J., Sayward, F., and Shaw, M. 1981. *Software Metrics: An Analysis and Evaluation.* MIT-Press.

Raymond, E. and Enterprises, T. 2000. The Cathedral and the Bazaar. *? ?*, 1–35.

SourceForge. 2010. SourceForge Web Site. http://sourceforge.net/.

Strein, D., Kratz, H., and L
"owe, W. 2006. Cross-language program analysis and refactoring. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation.* Citeseer, 207–216.

Tegarden, D., Sheetz, S., and Monarchi, D. 2002. Effectiveness of traditional software metrics for object-oriented systems. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on.* Vol. 4. IEEE, 359–368.

WANG, Y., LI, Q., CHEN, P., AND REN, C. 2007. Dynamic fan-in and fan-out metrics for program comprehension. *Journal of Shanghai University (English Edition) 11,* 5, 474–479.

YOURDON, E. AND CONSTANTINE, L. 1979. Structured design. Fundamentals of a discipline of computer program and systems design.