Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 233 (2009) 89–103
www.elsevier.com/locate/entcs

Quality Factors and Coding Standards – a Comparison Between Open Source Forges

Andrea Capiluppi, Cornelia Boldyreff, Karl Beecher

Centre of Research on Open Source Software – CROSS
Department of Computer Science
University of Lincoln, UK
Email: {acapiluppi,cboldyreff,kbeecher}@lincoln.ac.uk

Paul J. Adams

Research and Development,
Sirius Corporation Ltd.,
Weybridge, UK
Email: paul.adams@siriusit.co.uk

Abstract

Enforcing adherence to standards in software development in order to produce high quality software artefacts has long been recognised as best practice in traditional software engineering. In a distributed heterogeneous development environment such those found within the Open Source paradigm, coding standards are informally shared and adhered to by communities of loosely coupled developers. Following these standards could potentially lead to higher quality software.
This paper reports on the empirical analysis of two major forges where OSS projects are hosted. The first one, the KDE forge, provides a set of guidelines and coding standards in the form of a coding style that developers may conform to when producing the code source artefacts. The second studied forge, SourceForge, imposes no formal coding standards on developers. A sample of projects from these two forges has been analysed to detect whether the SourceForge sample, where no coding standards are reinforced, has a lower quality than the sample from KDE.
Results from this analysis form a complex picture; visually, all the selected metrics show a clear divide between the two forges, but from the statistical standpoint, clear distinctions cannot be drawn amongst these quality related measures in the two forge samples.

Keywords: Coding Standards, Open Source Software, Complexity.

1 Introduction

Programming styles and coding standards form a set of formal rules which are internally shared among programmers, and enforced by software managers [29]. These rules reflect different concerns and affect areas of source code writing, with the aim of improving both the readability of source code and the maintainability of the underlying software system. They range from language-independent *typographic styles*, as the rules affecting how both the source code and comments are visually

doi:10.1016/j.entcs.2009.02.063

structured and displayed [26,28], to general *programming practices* relative to specific programming languages (as C++ [16]), paradigms (such as the object-oriented paradigm [21]) or development approaches (such as the Agile methodologies [31]).

Research evidence has confirmed the presence of coding standards and programming styles in a wide spectrum of approaches: in commercial, proprietary, software [29], in Agile-driven systems [31] as well as in Open Source software [14]. Past empirical research also suggested that programming styles, applied in both creating new code or maintaining existing programs, have an impact on the quality of the resulting software [28].

In this paper it is argued that, in the presence of coding standards and programming guidelines, the resulting software will display a higher quality than software produced without such a coding framework. Specifically, a study of whether software produced within the Open Source paradigm has higher quality as long as they implement and enforce coding practices is provided. In order to detect the quality of the resulting software, three metrics were collected, both at the granular level of functions (or OO methods), the fan-in, the fan-out (both termed as "coupling") and the relative cyclomatic complexity. A null hypothesis was formulated: software projects from the sample implementing coding styles and standards will display lower values of complexity and coupling than counterparts from a sample not enforcing the same standards. The first sample ("with treatment") was randomly extracted from the KDE forge, already known for enforcing these standards and guidelines [14]. The other sample ("without treatment") was extracted from the SourceForge repository [1]. A statistical test [32] was used to evaluate the significance of the differences in the two samples.

This paper is structured as follows:Section 2 provides a description of past and current research works which can be related to the present paper, and describes how it completes or enhances previous approaches. Section 3 presents the definitions and the attributes used in this work. It also introduces and instantiates the GQM approach [2], tailoring a research question for the current study.Section 4 produces an empirical hypothesis based on the presented research question, illustrating the *null* and the *alternative* hypotheses, as well as the complexity and coupling metrics used to evaluate them. In the same section, an overview of the samples from the two forges is also given. Section 5 presents the results of the statistical tests linked to the hypothesis, and evaluates whether the null hypothesis should be rejected or not.Section 6 illustrates both internal and external threats to validity to this empirical study, whileSections 7 and 8 conclude this paper.

2 Related work and Background

The research areas related to this work can be divided into three main sections: metrics for software quality, literature on coding standards and styles and related practice including tool support. Regarding the first research area, there is an extensive

[1] Even if not clearly stated, it may very well the case that particular projects within SourceForge adhere to coding standards anyway. In the following, it is assumed that this effect is negligible.

software literature dealing with the measurement of software quality and associated characteristics such as complexity, comprehensibility, reusability and maintainability. Measurements have been traditionally divided into development and design quality metrics (early in the product lifecycle), e.g. [20], and post-release metrics which can be applied to a finished software product. One of the most famous effort in the later area presented the results of the impact of coupling, cohesion and complexity on the development cost of object-oriented systems [4].

Quality measurement has also matured to the point at which a standard has been defined for this activity. The ISO standard for software quality measurement [1] defines the characteristics of software quality as: functionality, reliability, usability, efficiency, maintainability and portability. In the past, there have been other attempts to determine quality numbers based on source code metrics: the most well-known is the Maintainability Index [27], which is a composite metric to assess, at the system level, the relative maintainability. The Halstead Effort [15], the McCabe cyclomatic complexity, the lines of code and comments are averaged into a single number to represent a global index of maintainability and quality. As reported in other studies [6], the McCabe values typically follow a Power-law distribution for methods or procedures in a system: using the average for such distributions will hinder some of the characteristics of the distribution, such as its skewness. The Halstead Effort metric has been evaluated as highly correlated with the McCabe index [17], but also heavily criticized as an unreliable metric [19]. In this work, the McCabe indexes are evaluated both as distribution in each project, and considering the fraction of highly complex functions (i.e., whose index is larger than 10 [23]). The Halstead Effort as a metric is not considered in this work.

Works focusing on open source quality or success often measure endogenous metrics (often "people" or "process" metrics) such as the amount of developer activity, the number of developers, forum activity, version control etc. [8,10,12,25]. A related work examining product metrics by Stamelos et al [30] undertook an analysis of 100 open source programs written for Linux and took a variety of basic static measurements (such as program length, unconditional jumps etc.). These results were compared to "industrial standards" suggested by the measurement tool.

The use of coding standards is also well covered in the literature and can be partially enforced through automated checkers, such as the UNIX-based *lint*, developed for the C programming language in the late 1970s [18]. Similar tools were later made commercially available for other programming languages, like the C++ [24]. Within the FLOSS community, there are lint-like tools available and the tool CQual++ is available for both C and C++. The use of CQual++ on Debian packages to improve their code quality has been recently reported [7] although it should be noted that this application has taken place *after the fact*, rather than as a regular practice by the project's developer community. Also, this tool is mainly focused on code improvement through the elimination of format string vulnerabilities. Within the KDE community, there is an explicit KDE Quality Team [2] and on its webpages, developers are guided towards information regarding Trolltech's Qt application frame-

[2] http://quality.kde.org/

work which is widely the core widget set within development of KDE applications. Coding standards can be automatically enforced by using Parasoft's Insure++ in conjunction with Qt [3].

Each KDE project has the freedom to develop its own specific Coding Style; projects are recommended to follow the kdelibs coding style [4]. There is a coding style for kdelibs that is supported by a vim script in *kdesdk/scripts/kde-devel-vim.vim* that helps developers to keep the coding style correct. In addition to defaulting to the kdelibs coding style it will automatically use the correct style for Solid and kdepim code. Developers can add rules for other projects via the SetCodingStyle function.

In addition to these quality-related policies in KDE, there is also an automated quality assurance tool called the English Breakfast Network (EBN) [5]. The EBN is a tool for detecting and measuring aspects of quality within KDE as a whole. For example, the EBN measures code defects and errors in source documentation, such as spelling errors.

3 Definitions and GQM approach

The following concepts and attributes have been used to extract data from the two OSS forges, and to define the complexity and quality attributes (*i.e.* coupling) used to compare the samples. These metrics have been evaluated at the finest granularity level (*i.e.*, source functions or methods); in order to compare them on the system level, they should be elevated to system level [5]. Since the McCabe complexity [22] was evaluated, and the aggregation of this metric at the system level is complicated by its own definition (*i.e.*, the number of independent paths of an executable function, plus one), the rest of this study will analyze the software quality at the function level.

(i) **Data points**: the selected KDE and SourceForge samples contain 50 projects each. The code of each project was downloaded from their respective Configuration Management System (either CVS or SVN) on the date of their latest available commit.

(ii) **Source Function**: at the smallest level of granularity, both the functions of the procedural languages, and the methods of the Object-Oriented paradigm are considered as source functions, and treated as the basic unit measure for this study.

(iii) **Complexity**: within the software engineering literature, this is a very broad term. To help in scoping it down, this paper consider complexity only from the point of view of the source functions. The attribute that will be used to empirically detect the complexity of source functions is the McCabe cyclomatic complexity. This is a measure of structural complexity, and it can be calculated

[3] http://www.parasoft.com/jsp/products/home.jsp?product=Insure
[4] http://techbase.kde.org/index.php?title=Policies
[5] English Breakfast Network: http://www.englishbreakfastnetwork.org

from a graph representation of a function, with each executable statement being a node on the graph, and arrows between the nodes showing the execution pathways and decision or branching nodes. Cyclomatic complexity is calculated as the number of decision (or branching) nodes plus one. In this research McCabe complexity (Mc) was evaluated for all the functions of each software system studied.

(iv) **Coupling**: it measures the degree to which each source function relies on other elements, that is, how interconnected is the code. Since this study is conducted at the function level, the union of all the function calls (and method invocations) form the network of couplings in a system. Each coupling can be uniquely categorized as inbound (c_in) or outbound (c_out) (or fan-in and fan-out), depending on the direction of the relative call. As an example, function 'sign off' (Figure 1) has two inbound (fan-in) and three outbound (fan-out) couplings. In this study we separately measured the number of inbound and outbound couplings of each function.

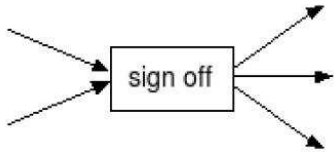

Fig. 1. example of 'inbound' and 'outbound' couplings for the function "sign off"

(v) **Instability**: from past literature, instability is the ratio of fan-out coupling (c_o) to total coupling (c_o + c_i) such that $I = \frac{c\_o}{(c\_o + c\_i)}$. This metric is an indicator of the resilience to change of software components (as in source functions) [13]. The range for this metric is 0 to 1, with $I = 0$ indicating the lowest instability for an element and $I = 1$ indicating a completely unstable element. In this paper it is argued that software systems should minimize the instability of its components in order to increase the quality of the underlying application.

3.1 KDE and SourceForge samples

The KDE forge [6] hosts a large number of OSS projects under a common name which together form both a desktop environment and associated application software, primarily for Unix-like operating systems. The KDE repository from which our samples are taken has somewhere in the order of 300 projects.

The SourceForge site [7] hosts more than 150,000 projects. However, it has been argued that a considerable proportion of SourceForge projects are to be considered as "tragedies" [10], owing to their failure to initiate a steady series of releases. No evidence of such a phenomenon in KDE is present, so to draw an accurate comparison, the sample from SourceForge has been extracted only from the pool of

[6] http://www.kde.org/
[7] http://sourceforge.net/index.php/

the "stable" projects (numbering approximately 20,000), *i.e.* those projects whose core developers labelled the status of the project with the tag "Production/Stable".

It was shown in a previous work [3] that certain exogenous characteristics of KDE differ significantly from SourceForge; specifically, KDE projects are, on average, subjected to a greater amount of activity from a greater number of developers and have been developed over a greater period of time. In addition, KDE projects were also observed to be significantly smaller than their counterparts within SourceForge.

3.2 GQM

The Goal-Question-Metric (GQM) method evaluates whether a goal has been reached, by associating that goal with questions that explain it from an operational point of view, and providing the basis for applying metrics to answer these questions [2]. The aim of the method is to determine the information and metrics needed to be able to draw conclusions on the achievement of the goal.

In the following, we applied the GQM method to first identify the overall goal of this research; we then formulate a number of questions related to the two OSS forges and their relative complexity (and conversely, their quality); and finally we collected adequate product and process metrics to determine whether the goal has been achieved.

(a) Goal: The long-term objective of this research is to assess the presence of coding standards within distributed environments, and to evaluate its effectiveness towards the creation of software artefacts. In particular, the goal is to investigate whether OSS forges which reinforce coding standards achieve higher quality software.

(b) Question: The purpose of this study is to establish differences between samples from KDE and SourceForge. Their complexity will be evaluated and a comparative research question will be evaluated via a direct comparison between the projects composing the two samples. Based on the complexity and coupling attributes defined above, the research question asks: in the presence of coding standards, will projects be less complex, and of higher quality, than of the counterpart forges not reinforcing any standards.

(c) Metrics: Every project from both samples will have their functions evaluated in terms of McCabe cyclomatic complexity (to assess structural complexity) and the fan-in and fan-out metrics (to assess the coupling). As a compounded metrics, the value of *instability* will also be evaluated based on the definition given above. Once completed, these measurements will be summarised per project for both metrics, specifically into a median value, a maximum value and a variance.

4 Research Hypotheses

Two related hypotheses have been formulated based on the research question composing the GQM approach. The objective is to show that the samples from the two forges achieve different levels of complexity and coupling, as measured by the three

attributes introduced above (mc, c_i, I). The directional tests will be also considered, in order to establish whether one forge achieves statistically higher quality results than the other: this means that a test will be run to check whether the KDE sample achieved a smaller level of complexity, or whether the KDE level of fan-in is larger than SourceForge. Based on common design principles, in fact, software designers and programmers should aim for low fan-out and high fan-in [11].

The statistical test have been used in this evaluation is the Wilcoxon test [32]: this is a non-parametric test, and the assumption of normality is not needed to run this test, as per the parametric family of tests. Nonparametric tests are more powerful in detecting population differences: since the normality assumptions are not clearly satisfied when considering the distribution of complexity and coupling attributes, the Wilcoxon tests were applied.

In this paper, three hypotheses were considered, as also summarized in Table 1:

(i) **Complexity:** the first hypothesis relates to the complexity, measured through the cyclomatic index. The null hypothesis assumes that the KDE and the SourceForge samples display different quality, i.e. different distributions of cyclomatic complexity. In addition, the directional test will check whether the KDE sample achieves a statistically smaller complexity than the SourceForge counterpart. The tests used will be both a simple and a directional Wilcoxon test: they will be applied on the median, mean, maximum and variance of the distribution of each observed project. These values are reported in the rightmost parts of Table 3 for the KDE sample, and Table 4 for the SourceForge sample.

	Null (H_0)	*Alternative (H_1)*	*Test*
complexity	KDE and SourceForge display similar levels of complexity	KDE and SourceForge display different levels	$mc_{kde} = mc_{sf}$
	KDE displays smaller complexity than SourceForge	KDE displays larger complexity than SourceForge	$mc_{kde} < mc_{sf}$
fan-in	KDE and SourceForge display similar levels of fan-in	KDE and SourceForge display different levels	$c\_i_{kde} = c\_i_{sf}$
	KDE displays larger fan-in than SourceForge	KDE displays smaller fan-in than SourceForge	$c\_i_{kde} > c\_i_{sf}$
instability	KDE and SourceForge display similar levels of instability	KDE and SourceForge display different levels	$I_{kde} = I_{sf}$
	KDE displays smaller instability than SourceForge	KDE displays larger instability than SourceForge	$I_{kde} < I_{sf}$

Table 1
Summary of the hypotheses to be tested

(ii) **Fan-in:** the second set of hypotheses relates to the inbound coupling (or fan-in), measured through the inbound connections of each source function. The null hypothesis assumes that the KDE and the SourceForge show different distributions of the fan-in attribute. In addition, the directional test will check

whether the KDE sample achieves a statistically higher fan-in than the SourceForge counterpart. The tests will be applied on the median, maximum and variance of the distribution of each project from KDE and SourceForge.

(iii) **Instability:** the last set of hypotheses considers the compounded metrics presented above, which takes into account both the fan-in and the fan-out of each function. As above, the null hypothesis assumes that the KDE and the SourceForge show different distributions of the instability attribute. The added directional test instead compares the KDE and the SourceForge samples to detect whether the KDE one achieved a statistically lower instability. The tests will be applied on the median and variance of the distribution of each project from KDE and SourceForge. These values are reported in the central parts of Table 3 for the KDE sample, and Table 4 for the SourceForge sample.

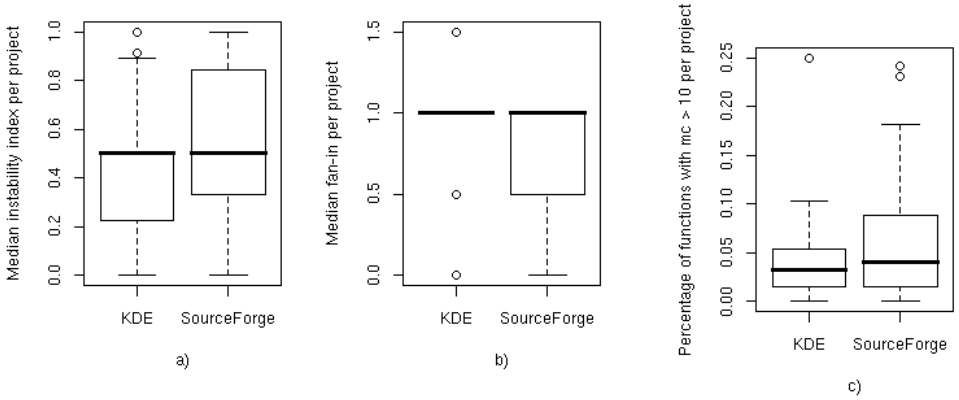

Fig. 2. Boxplots of measured attributes in the KDE and SourceForge samples

5 Results

This section summarizes the findings that were collected evaluating the research hypotheses. The attributes presented above were evaluated at the latest available change recorded in the CVS or SVN repositories: the couplings (fan-in and fan-out), instability and complexity of all the functions of the projects from KDE and SourceForge were calculated. For every project in the samples, the median and variance values of all the functions were used to compare the two forges. The maximum value was also used where appropriate.

As discussed above, a statistical test [32] allowed the null hypothesis to be either rejected or confirmed. The R programming language has been used to carry out these tests based on evaluation of the data extracted earlier from the respective forges [9]. A summary of the tests and their results is provided in Table 2 below: as visible, the majority of the paired tests have non-significant p-values, and therefore it is not possible to reject the null hypothesis.

Attribute	Metrics	W	p-value	
			1-tail	2-tail *a*: KDE > SF *b*: KDE < SF
Instability	median	693	*0.162*	*0.081*^{b}
	var	856	0.595	0.280^b
Fan-in	median	965	**0.032**	**0.016**^{a}
	max	590.5	**0.044**	**0.978**^{a}
	var	716.5	*0.058*	**0.9715**^{a}
McCabe	mean	767	0.73	0.36
	max	681.5	0.256	*0.128*
	var	872	0.49	0.247
	% $\geq$ 10	899	0.34	*0.17*

Table 2
Results of the Wilcoxon tests (p-values) when comparing the two forges – bold stands for high significance (95%), italics for weak significance

5.1 Complexity

In this work the complexity was measured, at the functions level, via the McCabe cyclomatic index [22]. As results, this paper reports on both the actual distribution of these indexes, and the conditions on the fringes of this metric. In particular, the percentage of the complexity indexes larger than a threshold (i.e., 10, from [23]) was evaluated for each project, in order to detect whether one forge achieves lower quality in terms of highly complex elements. Visually, the boxplots in figure 2 show a difference of distribution: it can be seen how the KDE forge has a much compact distribution of these highly complex functions, with only one outlier around 25%. On the other side, the SourceForge projects have a wider boxplot distribution, with a larger median, and two outliers in its distribution.

Statistically, and as visible from Table 2, only one difference was found to be significant: the median of the distributions of the complexity is statistically different in the two sample, and also the directional test is significant: the median of the complexities in the KDE sample is lower than the SourceForge counterpart. In all the other tests, this significance was not achieved, hence making it impossible to reject all the null hypotheses: from the data collected in the two sample, it is not possible to conclude that the KDE sample achieves a higher quality than SourceForge, from the point of view of the complexity achieved.

5.2 Coupling

In this work the coupling of a function is measured primarily by its instability. Whilst functions within KDE projects were observed to be more stable on average, they did not achieve a statistically significant difference from functions of SourceForge projects.

As visible from Table 2, the median value of fan-in for KDE projects proved significantly higher than that of SourceForge projects, with H_0 of hypothesis 1.2 rejected for the test $c\_i_{kde} < c\_i_{sf}$ at $p = 0.016$. However, the SourceForge projects (which are typically much larger than KDE projects) possessed significantly higher values of both variance and maximum coupling. While the projects in SourceForge may contain some functions that are used by a larger number of clients, any randomly chosen function from a KDE project is more likely to have a greater fan-in than that of a SourceForge project. However, examining in more detail the median fan-in results (summarized in figure 2) reveals the marginality of difference: in both forges, most projects have a median fan-in value of 1 (35 KDE projects and 29 SourceForge projects).

6 Threats to Validity

Two main threats to validity have been identified; they are as follows:

- Size of populations – The SourceForge sample was taken from a considerably larger population than the KDE forge: approximately 20,000 "production/stable" SourceForge projects, compared around 300 within KDE. Consequently, the static sample size of 50 is not the same proportion of the population in both cases.
- Automated coupling analysis – Because of the large amount of analysis necessary (100 real-world non-trivial software projects) the evaluation of coupling was automated by analysis software, which is presently at a level of sophistication that has the following consequences:
  - Test suites, when included within the software package, are included in the analysis and so contribute to the perceived level of fan-in. It is arguable whether or not test suites should be considered "part of the software";
  - The level of coupling, being measured by fan-in, is limited to being derived from a static view of the software; hence, dynamic coupling is not detected.

It is intended to refine the evaluation procedures for future analyzes to address these limitations in the analysis; and in future studies, the methods for selection of the sample populations will be given further consideration.

7 Conclusions

Within this paper the presence of coding standards and shared programming practices, and their effectiveness on the quality of the resulting software artefacts, were analyzed. In particular this work has argued that, within the Open Source paradigm, open collections of software projects could potentially benefit from higher

quality software by openly sharing documents and guidelines on programming styles and coding standards. In order to assess this, two OSS forges (collections of software projects) were selected: the first (KDE) is a large container of applications which share the same graphical interface objectives, and guidelines are shared among developers to comply with existing coding standards. The second (SourceForge) is a wider-spectrum collection of projects, and no explicit effort is attempted towards a common framework of coding rules.

Two similar sets of 50 projects were randomly selected from two similar pools of projects (*i.e.* the "stable" set of projects) of the selected forges. Two attributes were measured, as negative proxies for software quality, both at the level of granularity of functions (or methods): cyclomatic complexity and inbound coupling (fan-in). A third characteristic was also measured, based on the definition of instability, composing the inbound and outbound couplings. A Goal-Question-Metrics approach was applied, and a research hypothesis derived based on the measured characteristics: a random sample of sample projects from a forge (KDE) where active actions are taken towards quality will display more quality than projects were such effort is absent (SourceForge).

For the coupling characteristics of the software's functions (as measured by instability and fan-in) results showed only certain marginal improvements of KDE over SourceForge. Instability was, on average, lower in KDE projects than those of SourceForge, but not significantly so. Functions within SourceForge projects had a greater variance of fan-in (with upper values exceeding those of KDE), However median fan-in values of KDE projects appeared significantly larger than SourceForge projects. Only a more detailed inspection revealed that the improvement could likely be interpreted as marginal only.

For the complexity characteristics of the software's functions, results showed that graphical differences could be observed in the distributions of the highly complex elements, resulting in SourceForge being more complex than KDE. Statistically, it was not possible to conclude that a significant difference could be observed in the complexity characteristics of the two forges. A general result was extracted from the metrics collected: whereas some results proved satisfactory (e.g. the statistical significance of the fan-in tests), other results contradict (complexity) these findings, creating a multi-faceted vision of software quality.

8 Further Work

The most important aim for future work on this topic is to refine the metrics and include new ones where it would increase understanding of the property being measured. In particular it is intended to expand the scope of metrics examining modular structure, by quantifying a project's individual structurally complex parts (modules with fan-in and fan-out values over a certain threshold) and by investigating the dynamic coupling.

It is also intended to carry out new work in addition to that outlined here. Firstly, we will investigate the evolution of the software from the perspective of

these metrics by measuring their change over time and comparing the results from each forge. Secondly, having already identified a particular subset of the KDE forge that lays claim to high quality standards (the KDE PIM), it will be established if these claims are evidenced by any resultant effects on our measurements.

References

[1] *International Organization for Standardization. Software Engineering - Product Quality - Part 1: Quality Model. ISO, Geneva, Switzerland, 2001. ISO/IEC 9126-1:2001(E)* (2001).

[2] Basili, V. R., G. Caldiera and D. H. Rombach, *The goal question metric approach*, in: *Encyclopedia of Software Engineering* (1994), pp. 528–532, see also http://sdqweb.ipd.uka.de/wiki/GQM.

[3] Beecher, K., C. Boldyreff, A. Capiluppi and S. Rank, *Evolutionary success of open source software: an investigation into exogenous drivers*, Electronic Communications of the EASST: ERCIM Symposium on Software Evolution **8** (2007).

[4] Briand, L. C. and J. Wüst, *The impact of design properties on development cost in object-oriented systems*, in: *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics* (2001), p. 260.

[5] Capiluppi, A. and C. Boldyreff, *Identifying and improving reusability based on coupling patterns*, in: *ICSR '08: 10th International Conference on Software Reuse, Beijing China*, 25 - 29 May 2008.

[6] Capiluppi, A., A. E. Faria and J. F. Ramil, *Exploring the relationship between cumulative change and complexity in an open source system*, In CSMR '05: 9th Conference on Software Maintenance and Reengineering (2005), pp. 21–29.

[7] Chen, K. and D. Wagner, *Large-scale analysis of format string vulnerabilities in debian linux*, in: *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security* (2007), pp. 75–84.

[8] Crowston, K., J. Howison and H. Annabi, *Information systems success in free and open source software development: Theory and measures*, Software Process Improvement and Practice (2006), pp. 123–148.

[9] Dalgaard, P., "Introductory Statistics with R," Springer, 2002, 89–90 pp.

[10] English, R. and C. M. Schweik, *Identifying success and tragedy of floss commons: A preliminary classification of sourceforge.net projects*, in: *International Workshop on Emerging Trends in FLOSS Research and Development*, 2007.

[11] Ghezzi, C., M. Jazayeri and D. Mandrioli, "Fundamentals of Software Engineering," Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[12] Godfrey, M. W. and Q. Tu, *Evolution in open source software: A case study*, in: *ICSM*, 2000, pp. 131–142.

[13] Gorton, I. and L. Zhu, *Tool support for just-in-time architecture reconstruction and evaluation: an experience report*, in: *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 514–523.

[14] Halloran, T. J. and W. L. Scherlis, *High quality and open source software practices*, in: *Meeting Challenges and Surviving Success: 2nd ICSE Workshop on Open Source Software Engineering*, Orlando, FL, 2002.

[15] Halstead, M., "Elements of Software Science," Elsevier, New York, 1977.

[16] Henricson, M. and E. Nyquist, *Programming in C++, rules and recommendations*, Technical Report M 90 01 18 Uen (1992).
URL citeseer.ist.psu.edu/henricson92programming.html

[17] Henry, S., D. Kafura and K. Harris, *On the relationships among three software metrics*, SIGMETRICS Perform. Eval. Rev. **10** (1981), pp. 81–88.

[18] Johnson, S., *Lint, a c program checker*, in: *Unix Programmer's Manual, AT&T Bell Laboratories*, 1978.
URL citeseer.ist.psu.edu/johnson78lint.html

[19] Jones, C., *Software metrics: Good, bad and missing*, Computer **27** (1994), pp. 98–100.

[20] Kim, H. and C. Boldyreff, *Software metrics applicable to uml models: An investigation and an implementation*, in: *Proceedings of QAOOSE2002*, 2002.

[21] Kim, M., L. Bergman, T. Lau and D. Notkin, *An ethnographic study of copy and paste programming practices in oopl*, in: *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering* (2004), pp. 83–92.

[22] McCabe, T. J., *A complexity measure*, IEEE Trans. Software Eng. **2** (1976), pp. 308–320.

[23] McCabe, T. J. and C. W. Butler, *Design complexity measurement and testing*, Commun. ACM **32** (1989), pp. 1415–1425.

[24] Meyers, S. and M. Klaus, *A first look at c++ program analyzers*, Dr. Dobbs Journal **22** (1997).

[25] Mockus, A., R. Fielding and J. Herbsleb, *Two case studies of open source software development: Apache and mozilla*, ACM Trans. Softw. Eng. Methodol. **11** (2002), pp. 309–346.

[26] Mohan, A. and N. Gold, *Programming style changes in evolving source code*, iwpc **00** (2004), p. 236.

[27] Oman, P. and J. Hagemeister, *Construction and testing of polynomials predicting software maintainability*, Journal of Systems and Software **24** (1994), pp. 251–266.

[28] Oman, P. W. and C. R. Cook, *Typographic style is more than cosmetic*, Commun. ACM **33** (1990), pp. 506–520.

[29] Pfleeger, S. L., "Software engineering: the production of quality software," Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1987.

[30] Stamelos, I., L. Angelis, A. Oikonomou and G. Bleris, *Code quality analysis in open source software development*, Information Systems Journal (2002), pp. 43–60.

[31] Succi, G. and M. Marchesi, editors, "Extreme programming examined," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[32] Wilcoxon, F., *Individual comparisons by ranking methods*, Biometrics Bulletin **1** (1945), pp. 80–83.

		Instability (I)		McCabe (mc)			
	functions	median	var	mean	max	var	$McCabe \geq 10$
ark	341	0.5	0.16	3.73	62	43.79	8.5%
dolphin	725	0.5	0.2	2.08	37	6.11	1.7%
kaddressbook	1379	0.44	0.21	2.29	56	12.75	2.5%
kamera	87	0.5	0.21	3.24	24	19.98	5.7%
kate	2544	0.5	0.18	2.94	160	33.61	5.0%
kbattleship	479	0.5	0.19	1.69	17	2.06	0.4%
kdebugdialog	22	0	0.27	2.41	7	3.11	0.0%
kfind	131	0	0.21	2.53	55	27.1	1.5%
kgamma	49	0.42	0.2	3.55	25	19.59	8.2%
khangman	91	0	0.22	2.08	22	8.27	3.3%
khtml	10253	0.5	0.19	3.06	782	115.29	4.7%
kioclient	11	0	0.22	3.55	18	23.67	9.1%
kjs	2053	0.67	0.19	2.65	309	88.22	3.3%
kjsembed	1231	1	0.2	2.1	102	12.43	1.0%
klinkstatus	941	0.37	0.22	1.61	17	2.63	0.9%
kmag	107	0	0.24	2.53	32	13.36	2.8%
kmailcvt	116	0.89	0.2	3.74	17	15.38	10.3%
kmoon	37	0.25	0.18	3.11	43	51.38	5.4%
kmouth	340	0.33	0.18	2.53	17	6.55	2.9%
knetwalk	56	0	0.24	2.8	14	7.22	5.4%
knetworkconf	239	0	0.17	2.16	34	11.47	3.3%
knewsticker	67	0	0.26	1.78	16	4.15	1.5%
kpat	596	0.67	0.2	3.33	43	20.19	6.4%
kppp	716	0.5	0.16	2.49	84	18.94	3.6%
krfb	186	0	0.21	1.92	15	3.9	1.6%
krosspython	816	0.67	0.18	1.36	29	3.2	0.9%
ksim	1043	0.5	0.2	1.89	16	3.33	1.1%
ksquares	78	0.5	0.18	2.4	14	5.65	1.3%
kteatime	52	0.67	0.25	2.6	15	7.66	5.8%
ktnef	118	0	0.23	2.07	17	4.82	1.7%
kuiserver	102	0.5	0.22	1.89	21	9.03	3.9%
kxmlrpcclient	43	0.5	0.05	2	14	7.95	4.7%
lskat	242	0.5	0.18	2.82	25	11.62	5.4%
marble	1135	0.45	0.19	2.1	47	11.63	3.1%
nntp	20	1	0.2	7.3	25	34.96	25.0%
qtruby	1669	0	0.16	2.56	126	36.04	4.1%
shell	343	0.5	0.22	1.87	20	3.44	1.2%
solid	698	0.91	0.2	2.11	52	13.98	2.1%
sonnet	43	0.5	0.25	1.37	6	0.81	0.0%
umbrello	4683	0.63	0.17	3.21	94	30.83	6.6%

Table 3
Summary of instability and complexity attributes in sample KDE projects

		Instability (I)		McCabe (mc)			
	functions	median	var	mean	max	var	$McCabe \geq 10$
Aquila	48	0.75	0.25	3.81	38	36.92	8.3%
audiobookcutter	220	0.25	0.23	1.88	13	3.8	1.4%
Beobachter	223	0.5	0.22	1.53	9	1.09	0.0%
cotvnc	633	0.33	0.16	4.94	67	44.16	13.0%
cpia	370	0.5	0.12	8.36	96	203.26	24.1%
critical_care	2077	0.5	0.19	3.27	76	31.01	7.0%
csUnit	33	1	0.23	1.76	13	4.88	3.0%
expreval	327	0.42	0.22	1.72	51	15.23	1.2%
fitnesse	6944	1	0.18	1.31	40	1.07	0.1%
fn-javabot	492	0.25	0.23	2.48	43	14.54	4.1%
formproc	384	0	0.18	2.04	11	2.89	0.3%
fourever	1349	1	0.22	2.25	42	9.57	4.1%
freemind	2769	0.5	0.2	2.07	79	8.5	1.9%
galeon	4077	0.5	0.14	2.42	58	9.21	2.3%
hge	996	0.5	0.18	5.51	274	167.78	13.4%
icsDrone	33	0.5	0.11	7.48	108	346.82	18.2%
intermezzo	913	0.5	0.15	5.83	109	63.45	16.2%
j-trac	1154	0	0.22	1.55	19	2.32	0.9%
juel	824	1	0.19	1.93	69	9.98	1.7%
kpictorial	57	0	0.23	4.14	37	48.98	8.8%
mod-aspdotnet	93	0.5	0.21	2.8	18	9.97	6.5%
moses	10281	0.5	0.2	2.6	294	65.03	2.7%
nbcheckstyle	48	0.5	0.29	2.54	19	9.06	4.2%
neocrypt	68	0	0.26	2.96	16	12.64	8.8%
netstrain	26	0	0.13	5.04	23	30.52	23.1%
ozone	8288	0.89	0.2	1.96	65	5.64	1.6%
perpojo	201	0	0.21	1.58	9	1.82	0.0%
pf	58	0.75	0.17	2	66	127	24.1%
QPolymer	7039	1	0.19	1.2	65	2.54	0.6%
seagull	3256	0.5	0.2	5.86	344	183.12	14.4%
simplexml	70	0.33	0.13	5.53	26	31.38	14.3%
source	801	1	0.23	2.19	19	6.46	2.7%
swtjasperviewer	319	0.8	0.21	2.47	19	8.55	4.7%
txt2xml	165	0.63	0.2	1.65	10	2.03	1.2%
uniportio	42	0	0.17	3.5	23	18.79	4.8%
ustl	1801	0.6	0.16	1.43	16	1.24	0.3%
whiteboard	382	1	0.15	3.11	89	29	4.5%
wxactivex	259	0.5	0.19	2.53	36	18.93	5.0%
xmlnuke	127	1	0.2	2.47	20	8.7	3.9%
xqilla	4174	1	0.21	2.77	760	250.02	3.9%

Table 4
Summary of instability and complexity attributes in sample SourceForge projects