# *Code quality analysis in open source software development*

Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou & Georgios L. Bleris

Department of Informatics, Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece, email: stamelos@csd.auth.gr, lef@csd.auth.gr, bleris@csd.auth.gr

**Abstract.** *Proponents of open source style software development claim that better software is produced using this model compared with the traditional closed model. However, there is little empirical evidence in support of these claims. In this paper, we present the results of a pilot case study aiming: (a) to understand the implications of structural quality; and (b) to figure out the benefits of structural quality analysis of the code delivered by open source style development. To this end, we have measured quality characteristics of 100 applications written for Linux, using a software measurement tool, and compared the results with the industrial standard that is proposed by the tool. Another target of this case study was to investigate the issue of modularity in open source as this characteristic is being considered crucial by the proponents of open source for this type of software development. We have empirically assessed the relationship between the size of the application components and the delivered quality measured through user satisfaction. We have determined that, up to a certain extent, the average component size of an application is negatively related to the user satisfaction for this application.*

*Keywords*: Code quality characteristics, open source development, software measurement, structural code analysis, user satisfaction

## INTRODUCTION

Open source software development (referred to simply as *open source* in the following) is based on a relatively simple idea: the core of the system is developed locally by a single programmer or a team of programmers. A prototype system is released on the Internet, which other programmers can freely read, modify and redistribute the system's source code. The evolution of the system happens in an extremely rapid way; much faster than the typical rate of a 'closed' project.

Open source has managed to produce some impressive products such as the Linux operating system, the Apache Web Server and the Perl language. Netscape has also launched an

© 2002 Blackwell Science Ltd

open source project for Mozilla, its new Web Browser, proving that open source is a serious candidate for the development of large-scale commercial software. Overall, it appears that open source is presenting the traditional software development industry with an important challenge.

People of the 'open source community' claim that this evolutionary process, based on the combined expertise of an unlimited number of programmers/users, produces better software than the traditional closed model, in which just a single development team of a limited number of programmers has access and modification rights on the source code. Programmers involved in open source projects are highly motivated because they produce software mainly for personal satisfaction and, therefore, they are also expected to be highly productive.

Although most promising, there are various concerns regarding open source as a development philosophy that aims to produce high quality software systems. One issue is that the open source development process is not well defined (McConnell, 1999). The project is normally directed by the initial creator, who is responsible for any management activities (e.g. release of new versions, configuration management) of the rapidly changing new system. There are crucial development activities, such as system testing and documentation that are ignored. The requirements are defined by the programmers themselves. In practice, only general market requirements may be satisfied. Only detailed design seems to gain some attention, whereas most of the effort is definitely dedicated to coding and debugging.

The most well known attempt to informally define an open source process is Eric Raymond's 'The Cathedral & the Bazaar' paper (Raymond, 2000). In this paper, some common principles underlying the process are described. The most known principles are 'release early and release often' and 'given enough eyeballs all bugs are shallow'. These two principles largely define the power of open source: (a) rapid evolution so that many users/programmers may be given the opportunity to use the new system and modify it, and no time is spent in 'unnecessary' management activities; and (b) many programmers working at the same time on the same problem, increasing the probability of its solution.

Open source supporters describe this innovative software development process (Bollinger *et al*., 1999) as a very intensive spiral model (Boehm, 1988). However, it seems that no risk assessment is ever performed and no measurable goals are set during open source development, as would be required by Boehm's spiral model. Moreover, Bollinger and colleagues (Bollinger *et al*., 1999) point out that an important requirement for open source code is that it should be 'rigorously modular, self-contained and self explanatory', to allow development at remote sites. Another reason for obtaining high quality code from an open source project is the fact that the next step may be the maintenance of the open product to address vertical marketing requirements. In this case, a close project should probably be launched; system requirements should be more precise, and design and documentation demands should be more stringent, requiring high quality code to work on.

It is easily seen that the open source philosophy has both advantages and disadvantages,

and a series of articles (Bollinger *et al*., 1999; McConnell, 1999; O'Reilly, 1999; Wilson, 1999) has appeared recently, discussing the pros and cons of open source. Software experts and researchers, who are not convinced by open source's ability to produce quality systems, identify the unclear process, the late defect discovery and the lack of any empirical evidence as the most important problems (collected data concerning productivity and quality). Recently, Harrison (2001) has emphasised the need for empirical studies of open source by the software engineering research community.

In this paper, we focus on the last issue: no data have ever been published in support of the various claims in favour of or against open source. The reduction of defect correction costs has not been recorded systematically, and the necessary quality characteristics of the source code have not been demonstrated by any thorough data analysis. Only recently, a couple of case studies have attempted to quantify various aspects of the open source development. Mockus and colleagues (Mockus *et al*., 2000) have examined developer participation, core team size, productivity and defect density. Godfrey & Tu (2000) have studied the evolution rate of Linux kernel.

As McConnell (1999) points out, the open source projects should not be compared with the average closed project. They should be compared with the software development effectiveness achieved by leading-edge organizations that use a combination of practices to produce better quality and keep costs and schedules down. This assertion suggests that the internal quality of the delivered open source product should be compared with the quality levels required by the modern software industry.

As already mentioned, the core of open source activities happens at the code level. It is, therefore, reasonable to focus there, measuring and assessing the resulting product, i.e. the delivered code. The purpose of this article is to report and discuss the results of a pilot case that examines the quality of the source code delivered by open source development. Another purpose of our analysis is to identify structural metrics that may help in distinguishing more than one candidate versions of a software component when determining the contents of an open source release, in particular metrics related to component size. To this end, we measured 100 applications developed for Linux, using a software measurement tool. We have assessed the results according to the industrial standard proposed by the tool itself for benchmarking purposes. Initially, we have limited our analysis to the component level of the applications, planning to extend the case study to the architectural level as well.

In the following, we briefly review some papers that are related to our subject and which inspired our work. For a long time, programming style has been recognized to be directly related to certain program quality characteristics such as clarity and understandability (Berry & Meekings, 1985). Recently, Mengel & Yerramilli (1999) studied the quality of 90 C++ novice student programs through the static analysis of source code, using the same tool that has been used in our study. In another paper, Pighin & Zamolo (1997) analysed statically 350 000 lines of source code of industrial programs written in C. They managed to identify a discriminatory function that allowed them to predict module reliability classes based on structural characteristics of the code.

## MEASUREMENT AND ASSESSMENT OF OPEN SOURCE CODE

For our case study, we have used Logiscope® (Telelogic, 2000), a comprehensive set of tools able to perform, automatically, code measurement and comparison with user-defined programming standards. Moreover, Logiscope provides its own programming standard that is the result of empirical conclusions that came out after the analysis of millions of lines of industrial source code. The tool is used by several large organizations to control their programming process. A total of 70 companies in the telecommunications, automotive/transport, aerospace/defence, energy, process control and industry sectors are being reported to use the tool and the underlying methodology. Another example of large organization using a similar approach is the US National Security Agency (NSA) (Drake, 1996). NSA reported that, after 3 years of measurement and quality assurance activities, they collected and analysed results on some 25 million lines of code; a sample drawn from more than one billion lines of code. They compiled a set of measures, similar to the one used in our study, and they used it to 'promote high-quality processes where they matter most – at the code level'.

Using Logiscope®, we examined a sample of 100 C programs found in the SUSE Linux 6.0 release. The total size of the code examined is 606 095 physical lines of source code. The selection of the programs was made at random, so that the set includes a wide variety of applications, resulting in a representative subset of the applications. By the term 'application', we mean a program written for Linux; it can be a compiler, a utility function (mail, zip, . . .), a device driver, etc. The programs were analysed using the Telelogic Logiscope Code Checker and Viewer functions to calculate values of selected metrics and obtain recommendations for code improvement.

As mentioned above, we limited our analysis to the component level. A 'component' is any C function in the program. The quality of the programs was defined as the conformance to the accepted range of values, as set in Logiscope. Data from these programs were collected for 10 metrics to measure component quality [for a thorough discussion on structural metrics, see (Fenton & Pfleeger, 1997)]. The metrics, along with the acceptable range set by the tool given in parentheses, are the following:

**1** number of statements (N_STMTS): counts the average number of executable statements per component [1–50].
**2** cyclomatic complexity (VG): as defined by McCabe (1976). It is a metric based on graph theory that represents the number of linearly independent paths in a connected graph, in our case, the component control flow graph. It is considered an indicator of the effort needed to understand and test the component [1–15].
**3** maximum levels (MAX_LVLS): measures the maximum number of nestings in the control structure of a component. Excessive nesting reduces readability and testability of a component [1–5].
**4** number of paths (N_PATHS): counts the mean number of non-cyclic paths per component. It is another indicator of the number of tests necessary to test a component [1–80].

**5** unconditional jumps (UNCOND_J): counts the number of occurrences of GOTO. This type of statements contradicts the principles of structural programming for sequential control flow [0].
**6** comment frequency (COM_R): this is defined as the proportion of comment lines to executable statements [0.2–1].
**7** vocabulary frequency (VOC_F): defined by Halstead (1975) as the sum of the number of the unique operands, $n_1$, and operators, $n_2$, that are necessary for the definition of the program. This metric provides a different view of component size [1–4].
**8** program length (PR_LGTH): measures the program length as the sum of the number of occurrences of the unique operands and operators. This metric provides also another view of component size [3–350].
**9** average size (AVG_S): Measures the average statement size per component and is equal to PR_LGTH/N-STMTS [3–7].
**10** number of inputs/outputs (N_IO): counts the number of input and exit nodes of a component. This metric controls the conformance to another known principle of structured programming (only one input and one output is allowed) [2].

The tool, by default, measures each component and evaluates it against four basic criteria, namely *testability*, *simplicity*, *readability* and *self-descriptiveness*, using the measured values. The criteria are taken from an international standard concerning the subcharacteristics of software quality (International Standards Organization (ISO), 1991). This standard is used by numerous companies in the software industry (Fenton *et al*., 1995) and, despite criticisms, it is considered an important milestone in the development of software quality measurement. Moreover, the above criteria reflect adequately the quality characteristics that are desirable for open source code, as described in *Introduction*. Open source code should be testable to allow rapid evolution and simple enough to allow frequent modifications and extensions. Obviously, it should be also readable and self-descriptive to facilitate these activities.

The tool proposes specific recommendations for each software component and criterion. The recommendation levels are the following: ACCEPT, COMMENT, INSPECT, TEST, REWRITE. Not all of the recommendation levels are used for every criterion, e.g. COMMENT is not related to testability, which results from structural information. The tool determines the recommendation by examining the number of component measures that fall within the acceptable range. Each criterion is related to a specific subset of the 10 metrics given above (Table 1). For example, to assess testability, the conformance to the predefined ranges for metrics VG, MAX_LVLS, N_IO is examined. Notice that only seven out of the 10 metrics are actually used for component assessment. A possible outcome of the tool analysis may be 'the testability of component X is acceptable' or 'the readability of component Y requires that Y be rewritten'.

The above analysis produces eventually four recommendations per component. However, for open source code quality analysis, a global assessment for each component, combining all four criteria, would be preferable. To this purpose, we used an aggregation mechanism that is incorporated in the tool. For each criterion, a score is computed taking account of the related

**Table 1.** Relationship between quality criteria and metrics

| Criterion | Related metric |
|---|---|
| Testability | VG, MAX_LVLS, N_IO |
| Simplicity | VG, N_STMTS, AVG_S |
| Readability | VG, PR_LGTH, MAX_LVLS, AVG_S |
| Self descriptiveness | COM_R |

**Table 2.** Component classification according to the score obtained

| Recommendation | Min | Max |
|---|---|---|
| ACCEPTED | 90 | 100 |
| COMMENT | 80 | 90 |
| INSPECT | 50 | 80 |
| TEST | 30 | 50 |
| REWRITE | 0 | 30 |

metrics, each one with a different weight according to the programming language used. For example, the formula for testability is:

$$\text{testability} = 40 * \text{CVG} + 40 * \text{CMAX\_LVLS} + 20 * \text{CN\_IO},$$

in which CVG is a binary variable that stands for 'conformance to the predefined range for metric VG', taking value '1' when this assertion is true and '0' otherwise. CMAX_LVLS and CN_IO are defined in a similar way. Conformance for all three metrics would result to a score of 100. Another weighted-sum expression calculates the final component score from the scores obtained for each criterion.

Eventually, for each component a final score is calculated, ranging from 0 to 100, based on the four scores obtained for each one of the criteria mentioned above. A final score of between 90 and 100 signals an acceptable component. On the contrary, a final score of 30 or lower reveals a component that must be rewritten from scratch. A component should violate most of the metric ranges to be assigned the recommendation REWRITE. Intermediate scores lead to the other recommendations (Table 2).

The aggregation mechanism is entirely based on the weighted-sum approach, and the use of an arbitrary numerical scale for component assessment may be considered a limitation of Logiscope® (for the application of more elaborate aggregation procedures in software evaluation, see Morisio & Tsoukiàs, 1997 and Stamelos *et al*., 2000). All components examined by the tool should obtain the ACCEPT recommendation to be considered conformant with the level of quality required by modern software industry.

Each component of an application has been measured and the mean value of each metric has been calculated across an application. Descriptive statistics across all applications are reported in Table 3. For each metric, the minimum, maximum, mean, standard deviation and

**Table 3.** Statistical description of the 100 Linux applications' source code measurements (abbreviations explained in the text)

| | Minimum | Maximum | Mean | SD | Median |
|---|---|---|---|---|---|
| N_STMTS | 3.00 | 92.00 | 23.43 | 12.25 | 21.65 |
| VG | 1.00 | 35.00 | 7.70 | 4.28 | 7.58 |
| MAX_LVLS | 1.00 | 8.00 | 2.99 | 0.81 | 2.94 |
| N_PATHS | 1.00 | 32767.00 | 1266.34 | 3317.85 | 704.96 |
| UNCOND_J | 0.00 | 1.96 | 0.14 | 0.30 | 0.00 |
| COM_R | 0.00 | 1.32 | 0.11 | 0.15 | 0.08 |
| VOC_F | 1.50 | 9.90 | 2.75 | 0.93 | 2.67 |
| PR_LGTH | 18.00 | 516.00 | 133.38 | 73.17 | 122.82 |
| AVG_S | 3.68 | 14.96 | 6.35 | 1.58 | 5.96 |
| N_IO | 2.00 | 6.03 | 2.92 | 0.77 | 2.80 |

median values are given. In some cases, extremely varying values have been observed, but this is normal given the wealth of applications examined and the high number of people that have been involved in the development of the applications.

On average, the numbers for N_PATHS, UNCOND_J, COM_R and N_IO are outside the predefined ranges. The obtained result for N_PATHS seems to be the most distant from the allowed range, but this is probably as a result of poor definition and interpretation of the metric's impact on code quality. Nevertheless, the metric is not considered in the subsequent criteria analysis.

The average value for UNCOND_J is 0.14, not far from the ideal value (0). Moreover, in certain cases the use of GOTO statements is not prohibited, e.g. when we wish to implement exception handling in a component.

We consider the non-conformance of N_IO and COM_R metrics to be more alarming. On average, each analysed component has approximately three input–output nodes, one more than that which is expected by the structured programming principles. This may lead to the necessity for additional test cases and complicates component integration testing. The number of comments is also too low: on average one comment line appears for every 10 executable statements, indicating that in open source little care is taken for documenting the code produced.

For each application component, a single recommendation was obtained. Next, for each application, the percentage of components falling in each recommendation class was calculated. Table 4 provides the statistical description of the percentage of components allocated in the five recommendation classes. In one extreme case (a utility application), it was judged that all components should be rewritten. In four cases, all components were considered of acceptable quality.

Overall, the results suggest that the mean value of acceptable components is about 50%. On average, a percentage of 31% needs further comments, a percentage of 9% needs inspection, a percentage of 4% needs further testing and another 5–6% need to be completely rewrit-

**Table 4.** Statistical description of the component allocation within the five recommendation classes according to Logiscope® industrial standard

| | Minimum | Maximum | Mean | SD | Median |
|---|---|---|---|---|---|
| ACCEPT (%) | 0.00 | 100.00 | 50.18 | 18.65 | 48.37 |
| COMMENT (%) | 0.00 | 66.66 | 30.95 | 14.09 | 31.83 |
| INSPEC (%) | 0.00 | 50.00 | 8.55 | 8.50 | 7.65 |
| TEST (%) | 0.00 | 16.00 | 4.31 | 4.14 | 3.55 |
| REWRIT (%) | 0.00 | 100.00 | 5.57 | 10.73 | 3.20 |
| UNDEFI (%) | 0.00 | 7.69 | 0.42 | 1.29 | 0.00 |

ten. A small percentage of components has not been assessed by the tool because of some computational problems. Note that software organizations, applying programming standards based on software metrics, would require that all components should be considered acceptable.

## COMPONENT SIZE AND USER SATISFACTION

As mentioned above, modularity is being considered a crucial characteristic of the open source code. To investigate the relationship between component size and user satisfaction, we characterised the external quality of the LINUX applications by assigning a user satisfaction rating to each one of them. However, for this part of the study, we have considered only the applications considered stable by the release distributors. After removing an application with outlying values, the decreased sample size was 83 out of 100. We have used the metrics N_STMTS and PR_LGTH to represent the size of the components.

User satisfaction is probably the most straightforward way for measuring the external quality of a program. We have used a simple rating system based on a four-point defect severity scale followed by IBM (Jones, 1992). User satisfaction is measured through the following ordinal scale:

A user experiences only superficial errors at worst.
B all major program functions working but at least one minor function is disabled or incorrect.
C at least one major function is disabled or incorrect.
D program inoperable.

For our pilot study, we asked four experienced LINUX users to provide satisfaction ratings for every application out of the 83 considered, for which they felt they were familiar enough to provide a sound assessment. All four users had an experience of more than 2 years with various Linux applications. The users were also asked to take into account the usability of the applications. We received answers only for 71 applications, as none of the users considered he was familiar enough with the remaining 12 applications.

For the statistical analysis, each metric was considered a random numerical variable. As our aim was to detect differences among the rating levels of user satisfaction for each metric, we performed one-way analysis of variance (ANOVA) for every metric by the factor (categorical variable) USRSAT representing user satisfaction. In cases where the ANOVA showed some indication of differences among the levels of the factor, we further used *post hoc* tests for multiple comparisons to locate which levels (ratings) were in fact different. Besides the standard ANOVA, we used also non-parametric analysis of variance tests such as the Kruskal–Wallis (K–W) and the Jonckheere–Terpstra (J–T) tests. The normality of the numerical variables used in standard ANOVA was checked by the non-parametric Kolmogorov–Smirnov (K–S) test. The SPSS® statistical package was used for all statistical tests.

In general, we did not find any relationship between the majority of the metrics we have considered in this case study and user satisfaction. This fact does not invalidate these metrics, as their purpose is to capture the effort needed for code development activities and is not meant to foresee explicitly user satisfaction. However, we have detected indication of relationship between component size and user satisfaction. In the following, we present the results regarding only these metrics.

Number of statements (N_STMTS): The variable is normally distributed (K–S-test with $P = 0.759$). The ANOVA of N_STMTS by USRSAT gave $P = 0.036 < 0.05$, which means that there is an indication of significant difference among the ratings of the factor. The J–T non-parametric test also showed some difference ($P = 0.011 < 0.05$). The *post hoc* tests and the graphs in Figure 1 show that there is a significant difference in N_STMTS between applications rated as 'A' and those rated as 'C'. The merging of the ratings B, C and D into a single rating created a new grouping of the data in two samples for which ANOVA showed some difference of the corresponding mean values of N_STMTS ($P = 0.013 < 0.05$). This last difference can be seen in Figure 2.

Program length (PR_LGTH): This variable is also normally distributed (K-S-test with $P = 0.110$). The ANOVA by USRSAT gave $P = 0.011 < 0.05$, indicating significant difference among the ratings of the factor. The J–T non-parametric test verified the difference ($P = 0.011 < 0.05$). Again, merging of the ratings B, C and D created two groups differing in the corresponding mean values of N_STMTS (ANOVA with $P = 0.014 < 0.05$). The differences among the ratings appear in Figures 3 and 4.

## DISCUSSION

Our study revealed some interesting results that we are going to discuss in detail. However, we should remember that general conclusions may be drawn only from more extensive empirical studies that should involve a sufficiently large number of open source products of different application domains, programming languages, degree of success and popularity among users. In this paper, we are mostly interested in identifying the nature of the benefits that structural code analysis may bring to the open source community.

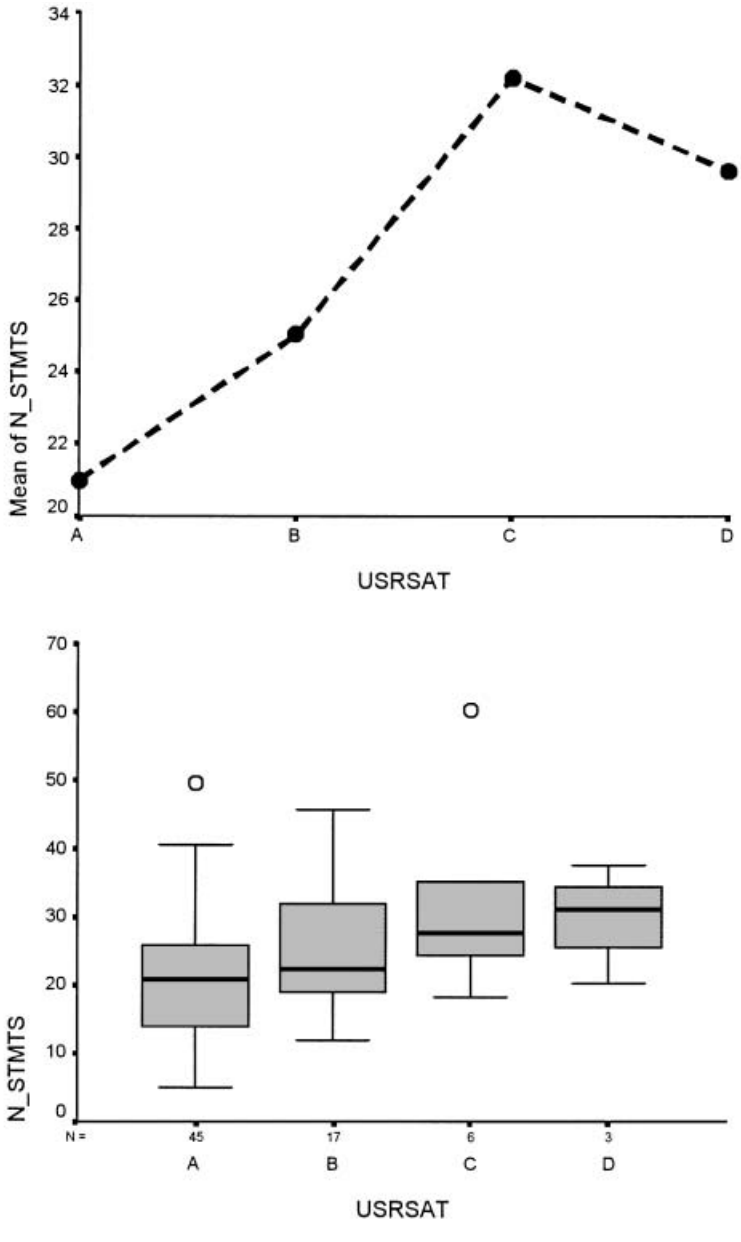


**Figure 1.** Differences in N_STMTS with respect to USRSAT.

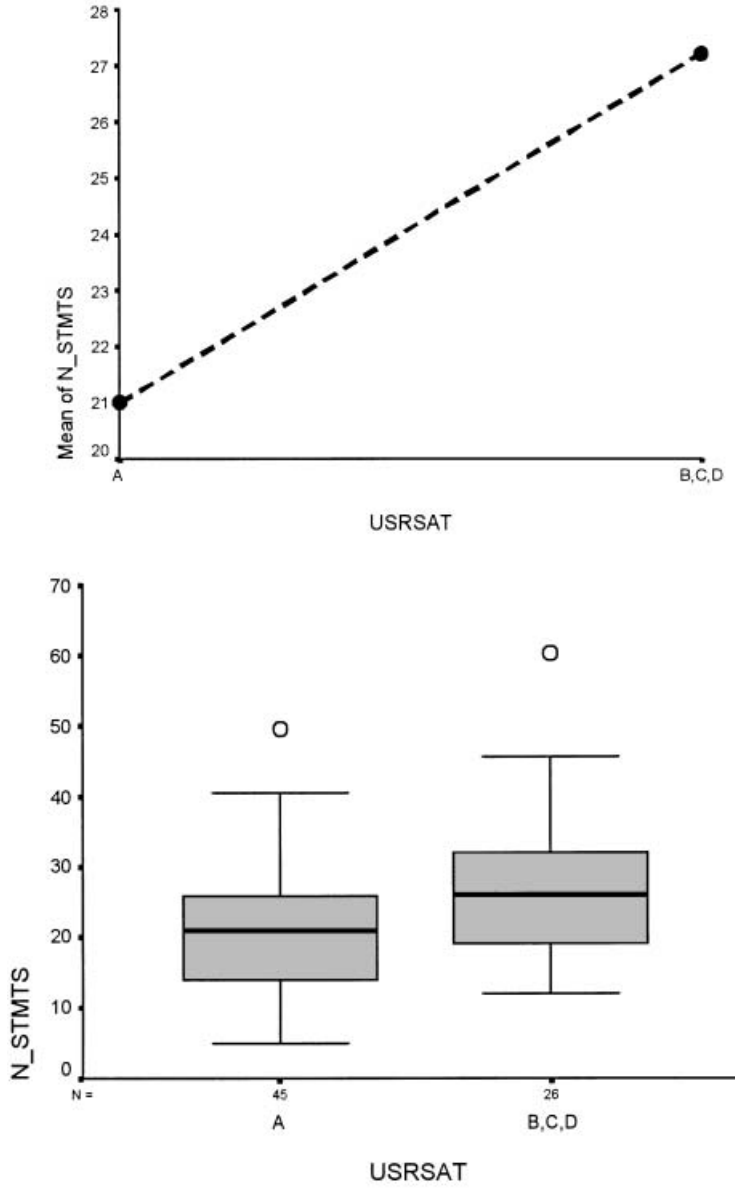


**Figure 2.** Differences in N_STMTS with respect to USRSAT (ratings B, C, and D are merged in one).

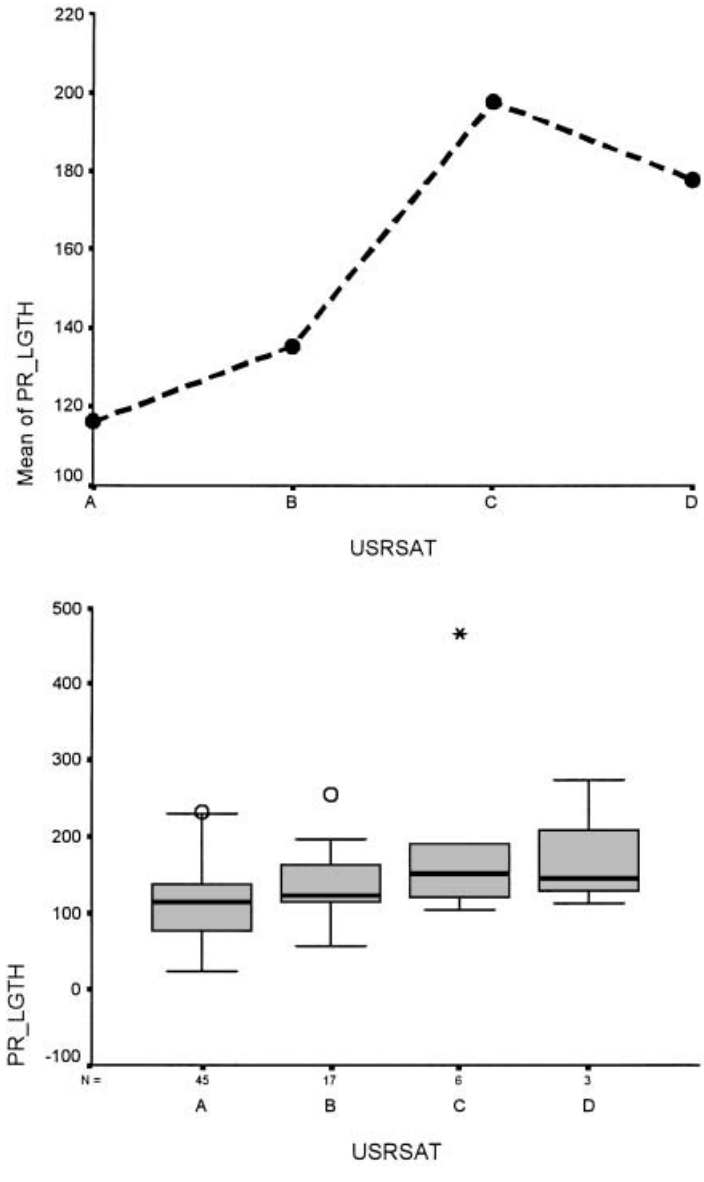


**Figure 3.** Differences in PR_LGTH with respect to USRSAT.

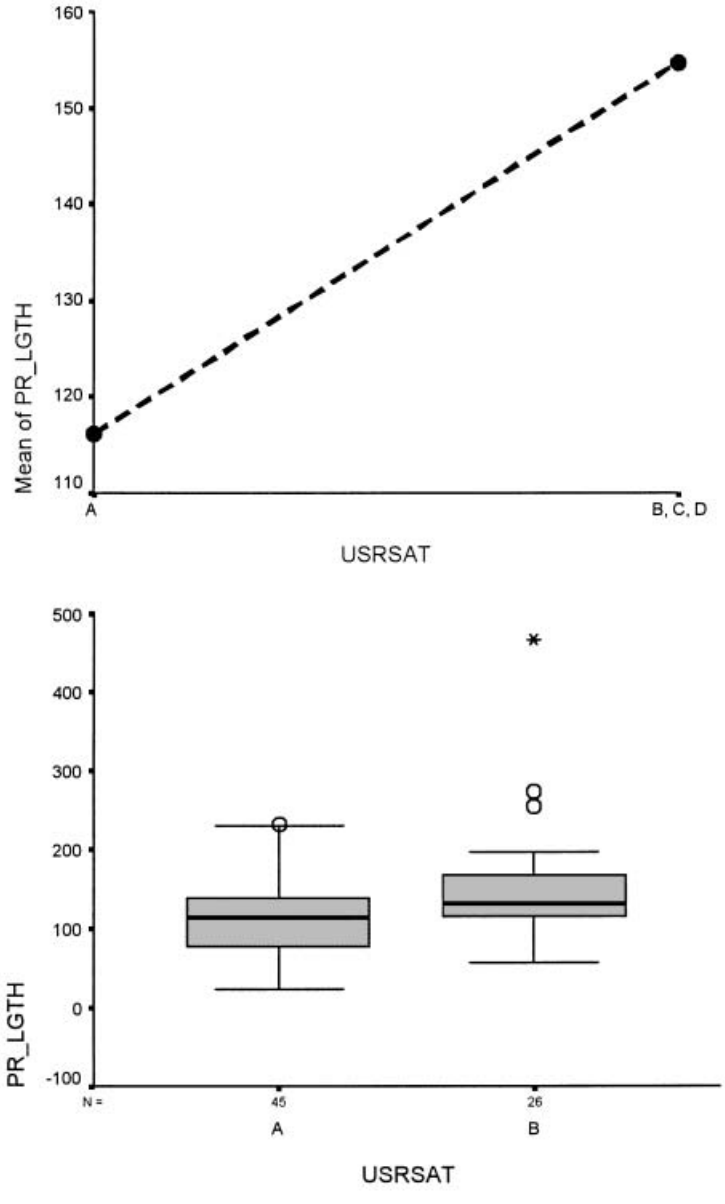


**Figure 4.** Differences in PR_LGTH with respect to USRSAT (ratings B, C, and D are merged in one).

According to our opinion, the results of the comparison with the industrial standard of Logiscope® may be interpreted as follows:

**1** the structural code quality of the Linux applications provides results higher than that which someone countering open source might expect, considering the limited control over the development process that has been followed.
**2** the structural code quality of the Linux applications provides results lower than the quality implied by the standard.

The first result is in favour of open source development. Traditional developers might fear that open source has a high probability of producing unreadable code, of low quality and impossible to maintain. They may think that open projects manage to survive only because a large number of programmers, with infinite patience because of their personal interest, are available to correct bugs and provide add-ons. From the data we have examined, it seems that this conjecture can not be supported. The average percentage of acceptable components across the programs is still high, as half of the components are in good shape. On the other hand, the average percentage of components that must be rewritten is not prohibitive for a corrective activity on the code. However, it should be noted that according to Pareto's law, a small percentage of the software will be responsible for the majority of the problems, so 5–6% of components needing to be rewritten is still a worrying result.

On the contrary, the second finding is against the open source development philosophy. Given the direct link between internal and external quality characteristics and the findings of this case study, it seems that the open source community should seriously take into account the need to develop higher quality code. This is suggested by the fact that, on average, almost half of the components of each application examined have not received the ACCEPT recommendation and must be reworked or revisited in some way (i.e. rewritten, tested, inspected or commented). Although the open source strength stems from the massive code-level peer review, such a suggestion implies that the way the code is structured necessitates even further work. Anyway, the quality of the code developed for Linux applications did not meet the requirements of the industrial standard we have considered, in contrast to what open source proponents have claimed up to now.

Looking back at the open source process it seems that structural code quality could be established as one of the project goals, defined as a set of structural measures and good programming rules. There are three distinct key practices that may help in achieving high quality code:

**1** the programmers could be asked to take into account structural code quality when intervening in the code.
**2** the project co-ordinator could assess the quality of the code returned by the programmers according to a predefined standard. This implies that certain components, non-conformant to the standard, may be rejected even if they provide correct bug fixes or new functionality.
**3** the project co-ordinator could take appropriate code re-engineering decisions whenever the project seems to experience severe problems.

According to our opinion the first practice is very hard to pursue because of the nature of the open source development. If not instructed otherwise by his professional environment, each individual programmer develops his own coding style, which most probably will not adhere to any coding standard at all. Even if the number of programmers following programming standards increases, it is not possible to guarantee that the new working version of a component will be produced by one of them. Besides all this, it is very tempting to violate coding rules of any kind when a quick solution may be achieved easily.

The second practice is not difficult, provided that project co-ordinators rely, not only on open source power, but on internal software quality as well. A programming standard must be followed, either based on a predefined industrial standard or based on systematic analysis of the code developed in similar projects. A measurement tool may then be applied before determining the contents of the new release. Modern tools of this type allow easy customization of the standard against which the code is measured.

We believe that the third practice is also a viable one. An issue with open source is that it gained popularity and respect because of a number of great successes. But what will happen in case a major open source project fails? The goal of structural code quality may be proved hard to respect while the project evolves rapidly. The project co-ordinator should rather monitor regularly the degree of code quality and should launch a re-engineering step at a certain point, in which, always following the open source style, the desired structural quality should be achieved. Such action may be necessary when a dead-end is reached when developing some component of particularly high complexity. This situation is analogous to what happens in an organization using a legacy system that can no long fulfil its mission. Although less attractive from a programmer point of view, the idea to re-engineer open source code might be necessary for open source to survive as a valid alternative to close development.

The analysis concerning user satisfaction demonstrated that, up to some extent, the average component size is negatively related to the external quality of the applications. In other words, applications with relatively small average component size seem to work better than applications that are composed of components of larger average size. This finding confirms the need for increased modularity already stressed by open source specialists (Bollinger *et al*., 1999). It is also contrary to what is known as the 'Goldilock's Conjecture', i.e. that there is an optimal module size, nor too small nor too big. They are also against the claims of other researchers (Hatton, 1997) stating that, 'in any software system, larger components are proportionally more reliable than smaller components'. As observed by Fenton & Neil (1999), if such statements were in generally true 'it would mean that program decomposition as a way of solving problems simply did not work', undermining fundamental concepts, like modularity, information-hiding, object-orientation and structured design.

Although there is indication of only a partial relationship between component size and user satisfaction, these findings could be still used when deciding the contents of an open source release. Release configuration is a frequent activity in open source development and might take place even daily (Raymond, 2000). When more than one candidate exists for a new version of an application, our analysis suggests that, all things being equal, the version characterized by the smaller component size could be selected for participation in the new release.

Intuitively, less component size is the result of better design and, consequently, of lower defect density and better user satisfaction. In addition, smaller component size should facilitate program maintenance and evolution. Given that this approach is easily automated, we believe that it can be easily integrated in the open source process.

## CONCLUSIONS AND FUTURE RESEARCH

In this paper, we have tried to provide empirical data to contribute quantitatively to the ongoing discussion concerning the real power of open source style of software development. Our goal was mainly to investigate the benefits that structural code analysis could provide to open source and provide clues for further empirical research. We have found that the quality of code produced in one case by open source is lower than that which is expected by an industrial standard, but not prohibitive of further improvement. We have speculated on a number of suggestions, quite common in industrial software development, aiming to enhance the evolving open source development process. We have also detected indication that increased modularity, measured as the average size of components in an application, is not only expected to facilitate open source development, but also is related to user satisfaction. Overall, we have imagined an open source process with the following features:

**1** the definition of a programming standard to be respected by the project participants at the launch of the project.
**2** the statical source code analysis in the stage before release content definition, to measure the code developed and verify conformance to the rules imposed.
**3** the utilization of the measurement results in the configuration of the new release.

Of course, the collected measurements might be further analysed to improve the performance of the project and provide guidance for new open source projects.

It is clear however, that more empirical evidence is needed to support any claims about open source quality. Source code analysis should be performed across more than one open source projects, considering architectural evaluation as well. The adoption of certain programming practices (heuristics) that are considered to improve code quality should also be investigated (Deligiannis *et al*., 2001). Open source may require the definition of its own quality standards, and each significant open source project might have custom code quality requirements, as is the standard practice with advanced software organizations. More research is also needed to correlate user satisfaction and its components (usability, functionality, etc.) to internal quality characteristics. Open source will also benefit from the definition of a more formal process that will allow the development of innovative supporting techniques and tools.

**ACKNOWLEDGEMENTS**

The authors would like to thank Telelogic for the use of Logiscope® at the University of Thessaloniki and Marco Mesturino for his helpful comments.

**REFERENCES**

Berry, R. & Meekings, B. (1985) A style analysis metric. *Communications of the ACM*, **28** (1), 80–88.

Boehm, B. (1988) A spiral model for software development and enhancement. *IEEE Computer*, **21** (5), 61–72.

Bollinger, T., Nelson, R., Self, K. & Turnbull, S. (1999) Open source methods: peering through the clutter. *IEEE Software*, **16** (4), 8–11.

Deligiannis, I., Shepperd, M., Roumeliotis, M. & Stamelos, I.. (2001) An empirical investigation of object-oriented design heuristics for maintainability. *Journal of Systems and Software,* (in press).

Drake, T. (1996) Measuring software quality: a case study. *IEEE Computer*, **29** (11), 78–87.

Fenton, N., Iizuka, Y. & Whitty, R. (eds). (1995) *Software Quality Assurance and Measurement: A Worldwide Perspective.* International Thomson Computer Press, London.

Fenton, N. & Pfleeger, S.L. (1997) *Software Metrics: a Rigorous and Practical Approach.* 2nd edn. International Thomson Computer Press, London.

Fenton, N. & Neil, M. (1999) A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, **25** (5), 675–689.

Godfrey, M. & Tu, Q. (2000) Evolution in open source software: a case study. *Proceedings IEEE International Conference on Software Maintenance.*

Halstead, M. (1975) *Elements of Software Science*. Elsevier, North-Holland.

Harrison, W. (2001) Editorial: Open Source and Empirical Software Engineering. *Empirical Software Engineering,* **6**, 193–194.

Hatton, L. (1997) Re-examining the fault density-component size connection. *IEEE Software*, **14** (2), 89–98.

International Standards Organisation (1991) *Information Technology-Software Product Evaluation: Quality Characteristics and Guidelines for their Use.* ISO/IEC IS 9126, Geneva.

Jones, C. (1992). *Applied Software Measurement*. McGraw-Hill, New York.

McCabe, T. (1976) A complexity measure. *IEEE Transactions on Software Engineering*, **2** (4), 308–320.

McConnell, S. (1999) Open source methodology: ready for prime time? *IEEE Software*, **16** (4), 6–8.

Mengel, S. & Yerramilli, V. (1999) A case study of the static analysis of the quality of Novice student programs. *Proceedings SIGCSE '99*, 78–82.

Mockus, A., Fielding, R. & Herbsleb, J. (2000) A case study of open source software development: the Apache Server. *Proceedings of the International Conference on Software Engineering.*

Morisio, M. & Tsoukiàs, A. (1997) IusWare, A methodology for the evaluation and selection of software products. *IEEE Proceedings on Software Engineering*, **144**, 162–174.

O'Reilly, T. (1999) Lessons from open source software development. *Communications of the ACM*, **42** (4), 33–37.

Pighin, M. & Zamolo, R. (1997) A predictive metric based on discriminant statistical analysis. *Proceedings ACM ICSE '97*, 262–269.

Raymond, E. (2000) *The Cathedral and the Bazaar. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/*.

Stamelos, I., Vlahavas, I., Refanidis, I. & Tsoukias, A. (2000) Knowledge-based evaluation of software systems: a case-study. *Information and Software Technology*, **42** (5), 333–345.

Telelogic (2000) *Logiscope User's Manual, V3.1*. Telelogic, Paris.

Wilson, G. (1999) Is the open source community setting a bad example? *IEEE Software*, **16** (1), 23–25.

## Biographies

**Dr Ioannis Stamelos** has been a Lecturer of Computer Science at the Department of Informatics, Aristotle University of Thessaloniki, Greece, since 1997. He received a BSc degree in Electrical Engineering from the

Polytechnic School of Thessaloniki and a PhD in Computer Science from the Aristotle University of Thessaloniki (1988). He worked as a Senior Researcher at Telecom Italia from 1988–1994, and as a Systems Integration Director at STET Hellas, a mobile telecom operator, from 1995–1996. He teaches courses on language theory, object orientation, software engineering and information systems. His research interests include evaluation, cost estimation and management in the areas of information systems and open source software. He is a member of the IEEE and IEEE Computer Society.

**Dr Lefteris Angelis** received both his BSc and PhD degrees in Mathematics from the Aristotle University of Thessaloniki. He is specialized in the study and combinatorial construction of optimal experimental designs and, more generally, in statistical methods. He has worked since 1999 as a Lecturer at the Department of Informatics of Aristotle University, teaching courses on calculus and applied mathematics. His research interests involve algorithmic and combinatorial planning of optimal experiments, computational methods in mathematics and statistics, random search algorithms and statistical methods with applications in various areas of software engineering and particularly in the estimation of cost of software projects.

**Apostolos Oikonomou** holds a BSc degree in Computer Science from the Aristotle University of Thessaloniki. He is now pursuing a MSc degree on Information Systems at the Imperial University, London, UK. He is interested in software measurement, CASE tools and open source development.

**Prof. George L. Bleris** is a Professor of Mathematics at the Department of Informatics, Aristotle University of Thessaloniki, Greece. He received a BSc degree in Mathematics, and a PhD in Physics from the Aristotle University of Thessaloniki (1980). He has served as a Professor or Assistant Professor in the Departments of Physics and Informatics for 20 years. He teaches courses on physics, thermodynamics, statistical physics, group theory, linear algebra and discrete mathematics. Some of his research interests are cellular automata, non linear systems and information systems cryptography. He is the author of seven books and 70 scientific papers. Prof. Bleris is the Director of the Programming Languages and Software Engineering Laboratory at the Department of Informatics.