Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 233 (2009) 5–28
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Evaluating the Quality of Open Source Software

Diomidis Spinellis[1] Georgios Gousios[1] Vassilios Karakoidas[1]
Panagiotis Louridas[1]

Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece

Paul J. Adams[1]

Research and Development
Sirius Corporation Ltd.
Weybridge, United Kingdom

Ioannis Samoladas[1] Ioannis Stamelos[1]

Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece

Abstract

Traditionally, research on quality attributes was either kept under wraps within the organization that performed it, or carried out by outsiders using narrow, black-box techniques. The emergence of open source software has changed this picture allowing us to evaluate both software products and the processes that yield them. Thus, the software source code and the associated data stored in the version control system, the bug tracking databases, the mailing lists, and the wikis allow us to evaluate quality in a transparent way. Even better, the large number of (often competing) open source projects makes it possible to contrast the quality of comparable systems serving the same domain. Furthermore, by combining historical source code snapshots with significant events, such as bug discoveries and fixes, we can further dig into the causes and effects of problems. Here we present motivating examples, tools, and techniques that can be used to evaluate the quality of open source (and by extension also proprietary) software.

Keywords: open source, product quality attributes, process quality attributes, SQO-OSS

[1] This work was funded by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software (SQO-OSS)"

doi:10.1016/j.entcs.2009.02.058

1 Introduction

Traditionally, research on software quality attributes was either kept under wraps within the organization that performed it [4, pp. vii–viii], or it was carried out by outsiders using narrow, black-box techniques [19,6]. The emergence of open source software has changed this picture [31] by allowing us to examine both the software products [27] and the processes that yield them [13]. Thus, assets, such as the software source code, the associated data stored in the version control system, the issue-tracking databases, the mailing lists, and the wikis, allow us to evaluate quality in a transparent way [10]. More importantly, because open source software has considerable economic impact [8], and is increasingly used in mission-critical real-world applications (see for instance [5, p. 313] and [17, p. 81]), many organizations would like to have at hand object measures regarding the quality of the development process and the corresponding product.

This paper presents a technical and research overview of SQO-OSS, a cooperative research effort aiming to establish a software quality observatory for open source software. After an overview in the next Section, Section 3 presents the system's structure, and Section 4 examples of research on software quality that we hope to bring under the SQO-OSS umbrella.

2 Overview

The motivation behind this study came about three years ago when one of its authors (Spinellis) found on his hands an idle server with ample storage and internet bandwidth. Having recently read studies concerning the use of the maintainability index on open source quality [32,24], he decided to apply it on snapshots of the FreeBSD system over 10 years of its evolution.

The maintainability index (MI) is a widely used measurement of maintainability. Typical values for MI range from 200 to -100. Higher MI values imply better maintainability. The formula and its constituent coefficients are derived from numerous empirical studies, and the formula's results have been tested against actual programmer perceptions. For example, one study [3] relates how Hewlett-Packard (HP) engineers compared two similar systems. The system they subjectively considered as being difficult to maintain and modify had an MI of 89, the other, which had received praise for its quality in an internal HP evaluation had an MI of 123. Normally, we should calibrate the formula's coefficients for our specific organization and project, but even with its given values, the formula typically yields usable results.

To experiment with the formula, Spinellis put together a script to calculate the value over a directory tree, and then applied it on snapshots of the FreeBSD system over 10 years of its evolution (see Figure 1). That demonstrated the value of exploring quality attributes using process-related data, and initiated a research grant application, for SQO-OSS: a software quality observatory for open source software.

The SQO-OSS project is a two year 200 person-month European research effort.

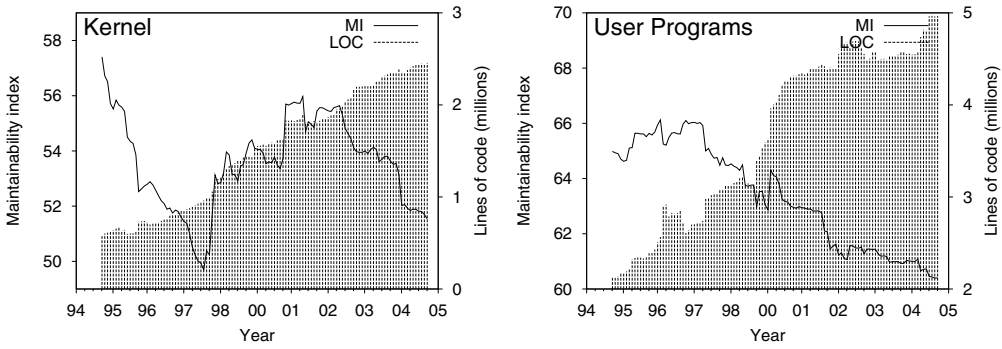


Fig. 1. Program growth and maintainability index over time in the FreeBSD kernel and user programs.

The consortium consists of two academic partners: the Athens University of Economics and Business and the Aristotle University of Thessanoniki, and three industrial partners: KDAB, with employees in Sweden, France, Denmark, and Germany, which executes Open Source and proprietary development contracts in the Qt/KDE environment; ProSyst, a leading provider of embedded Java and OSGi compliant software, and Sirius Corporation, an Open Source consultancy. Also participating is KDE e.V., an open source organization behind the namesake powerful free software graphical desktop environment for Linux and Unix workstations. The project's goals are to

- create a metric plugin-based architecture and a corresponding processing engine,
- establish new product and process software metrics that take advantage of the SQO-OSS infrastructure,
- provide an interface through the web, web services, and an Eclipse plugin that developers can use to improve the quality of their application,
- publish concrete values of product and process metrics for popular OSS software,
- setup a league of open source software applications based on user-specified criteria.

Each one of the above goals is not a unique or an innovative tool idea. There are several open source tools that try to evaluate code quality of a single software project by examining several aspects of it. PMD [1] is a Java scanner that tries to find possible bugs from exception handling statements and code problems, such as dead or duplicate code. Findbugs [2] performs static analysis to reveal bugs in Java based programs. Checkstyle [3] is a coding style checker for Java programs. Sonar [4], unlike the above, is a plug in metrics tool, for Java. It integrates, as plug-ins, a set of code measurement tools (like the ones presented) in a single application and presents overall results. The presentation follows the ISO/IEC 9126 Quality Model

[1] http://pmd.sourceforge.net
[2] http://findbugs.sourceforge.net
[3] http://checkstyle.sourceforge.net/
[4] http://sonar.codehaus.org/

[12]. A notable open source metrics collection tool for C++ is ESx[5] from National Research Council of Canada.

In addition to code quality assessment tools for standalone projects there are also frameworks that evaluate a fair amount of open source projects. These frameworks present their results on the web. Ohloh[6] performs measurements regarding the source code repository of almost 14000 open source projects. Metrics include lines of code, programming languages that are used by a project, developer contributions and licenses used. Apart from presenting numbers, Ohloh also makes estimations and statements about the development status, such as "Project X has and active team, an established codebase and an increasing development activity". Similar to Ohloh is Sourcekibitzer[7] which analyzes about 700 open source, Java based, projects. Sourcekibitzer presents reports like lines of code evolution and complexity evolution. Moreover, Sourcekibitzer also tries to evaluate developers' contribution to open source projects and has constructed a "developer know how" metric to measure developer's breadth of knowledge according to the participating open source projects. Scan by Coverity[8] is a collaboration between Coverity and the US Department of Homeland Security to apply Coverity's commercial static analysis tools to more than 14000 individual open source projects. They present their results on their website and also categorize projects to "rungs" according to their performance to their tests.

A key difference between SQO-OSS and these systems is its ability to calculate and integrate metrics from various product and process-related sources. These can include the actual code, actions on the version control system, posts in mailing lists, and entries in the system's bug database. Thus, SQO-OSS tries to take into account the open source development as a whole, not only code. Using a plug-in based system, SQO-OSS integrates various tools to gather measurements. Then these measurements are used to perform quality evaluation. In addition, SQO-OSS is not limited to a single programming language, but can apply appropriate metrics to programs in different languages. Its ambition to build an on line database of metrics of open source projects combines the features both from stand alone tools and the frameworks presented above. Unlike the frameworks presented SQO-OSS will be an open source project itself, open to the community and free for everyone to try and participate.

3 System Architecture and Implementation

The software produced by the SQO-OSS project is called Alitheia, for "neat and businesslike truth" (Alitheia means truth in Greek). A complete Alitheia deployment consists of a data collection system, a computation component called the cruncher, and a presentation layer in the form of a website. The data collection

[5] http://www.psmsc.com/ESx.asp
[6] http://www.ohloh.net
[7] http://www.sourcekibitzer.org
[8] http://scan.coverity.com

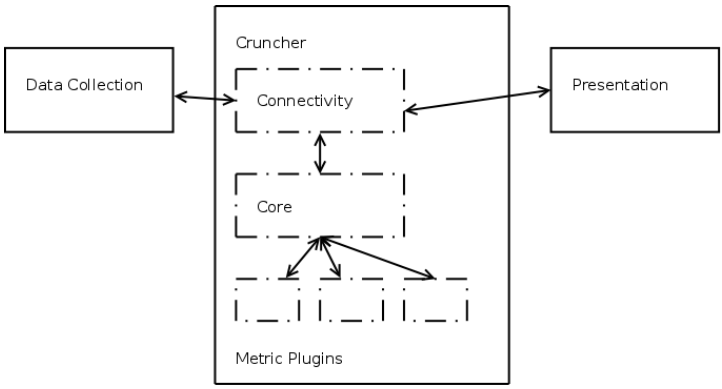


Fig. 2. Sketch of the architecture of an Alitheia deployment.

system collects the raw data from open source projects and the presentation layer makes computation results available to users; neither is relevant to this paper and the whole may be viewed as a standard three-tier architecture.

The cruncher part of the Alitheia system is a more complex artifact, as it brings together data storage, multi-level caching, metadata extraction and preprocessing as well as resource management for the actual computation work.

The cruncher is built on the OSGi framework (formerly the Open Services Gateway initiative). This framework manages loosely-coupled collections of components and provides lifecycle and remote management. This means that parts of the system may be selectively replaced in the field without affecting the rest of the system. It provides additional protection to system components through a strict separation of the different modules within the system. The cruncher consists of the actual computation core (which in turn handles caching, resource management, scheduling and some data storage), connection services for the other tiers of a complete Alitheia deployment and plugins that implement the computation of specific quality metrics such as the CLMT or MDE metrics described later in this paper. A sketch of the components is provided in Figure 2.

The core of the cruncher is a single module for OSGi, although it fulfils a number of separate roles. This monolithic (local) design was chosen to ease testing and performance issues; the components of the core are tightly coupled and we deemed that they cannot or should no be updated separately. The connection layer contains a Java servlet container for web-services and other connectivity. The portion of the system that sees the biggest benefit from the OSGi framework is the collection of metric plugins that may be extended, disabled, removed and upgraded through a combination of cruncher functions (removing local data storage for a metric plugin) and OSGi functions (unloading the code of a plugin, for instance).

To illustrate how some of the components of an Alitheia deployment work together, we walk through a typical sequence of events that trigger computation and storage within the cruncher. To do so, we must start with an external source: an open source project that is being studied or monitored by the Alitheia deployment. Suppose a developer changes some of the code of this project and commits the

change. Then the following events occur within the Alitheia deployment:

(i) Some time later, the data collection component updates its copy of the open source project and notices that the source code has changed.

(ii) The data collection component updates the raw data from the data source and connects to the cruncher to inform it of the change in the raw data.

(iii) The cruncher retrieves the raw data from the data collection component, analyses it and stores the extracted and preprocessed metadata locally.

(iv) The cruncher determines which metric plugins should act on the new data. Each of these plugins is asked to calculate a result for the change(s) in the raw data.

(v) The scheduler part of the cruncher handles resource and CPU allocation to the computation jobs that ensue.

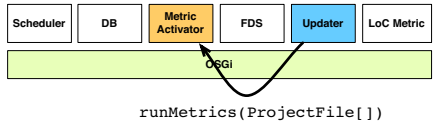
(vi) Each metric plugin does its calculation and stores its results.

Once the core has activated metric plugins for calculations, the roles of master and servant are reversed: the metric plugins begin querying the core for services. The core provides two levels of data access, each with their own caching scheme, through a Thin and a Fat Data Services Layer. Metrics may use either layer but the Fat Layer is recommended, as it provides more processed and cached data than the Thin Layer.

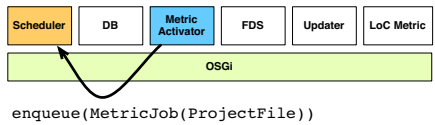
The Thin Data Layer (TDS, so called because using it for data retrieval is a tiresome process) provides raw project data to clients (e.g., metrics plugins). The raw data consists of project source code, both as individual file contents and source checkouts, project source history, mail messages in RFC822 format and bug data. The TDS manages access to the data and does resource management so that raw data requests do not overwhelm the cruncher (for instance by simultaneously requesting a complete checkout of the source code of KDE for all 830,000 revisions of that project).

The Fat Data System (FDS) deals with the processed metadata about individual items that would otherwise be retrieved through the TDS; for mail messages we may consider sender, recipients, subject, etc. bits of metadata that can be individually queried. The FDS also performs aggregation and allows higher-level search: "which mail messages were sent last tuesday?" or "what replies are there to this message". The FDS uses the database storage for the cruncher to store the metadata. We assume that metadata is both smaller and used more briefly than raw project data. Another feature of the FDS is the production of "timeline" views of a project, in which it merges the events from the source, mail and bug data into a single unified notion of "project change event." This is valuable for metrics that operate on more than one datatype or that attempt to measure an aspect of a project's *process*, not just the product.

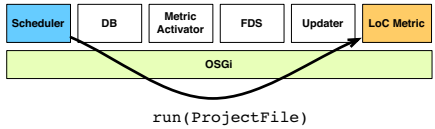
From the point of view of a metric plugin — once that metric is activated to do a specific measurement — the architecture of the Alitheia core is turned on its head: the FDS is the primary service to use, with the database of metadata directly



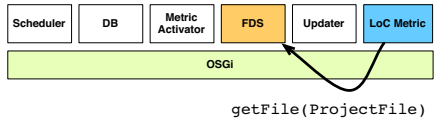
(a) The updater notifies the metric activator



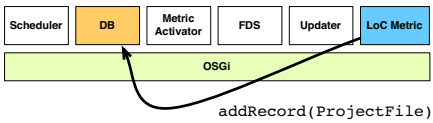
(b) The metric activator enqueues a calculation job



(c) The job is executed



(d) The metric retrieves the file contents from the FDS



(f) The result is stored

Fig. 3. Metric activation and processing

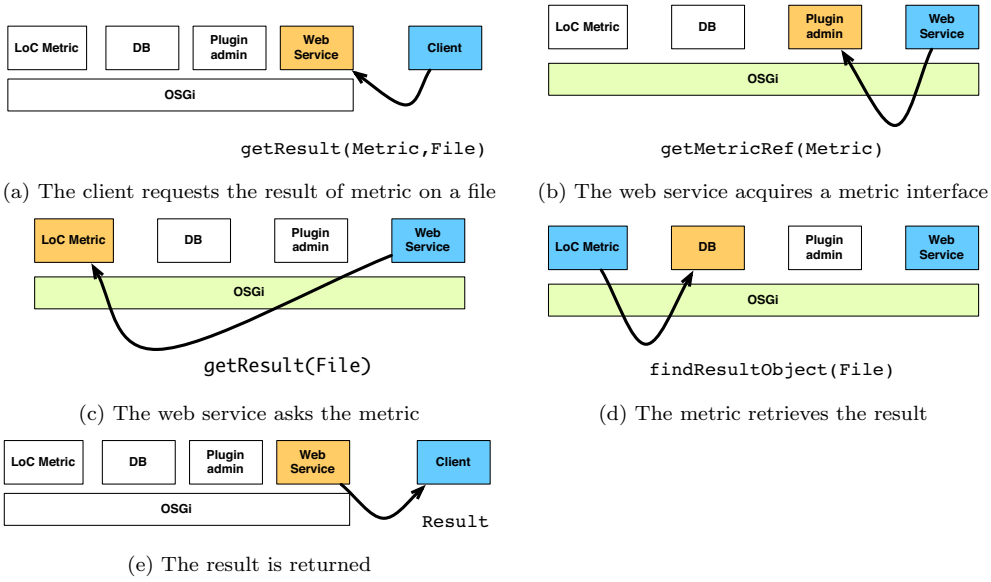
available if the FDS API is insufficient, and the TDS is to be used for low-level data shuffling; the metric's primary concern is in obtaining the data and storing its result. Overall, a metric may view the rest of an Alitheia deployment as an elaborate multi-level cache mechanism, where remote data from an open source project under study by the metric is mirrored by the data collection subsystem (reducing latency for raw data access), then copied to the cruncher for immediate study through the TDS (reducing data access time further, but still requiring processing to obtain the common metadata, if that is needed) and stored in pre-digested form (reducing the time to obtain common metadata further) in the FDS.

Turning the view on its head again and examining the interface that a metric provides to the cruncher, we see that this interface has three areas of functionality: lifecycle management, measurement (both performing measurements and obtaining the results afterwards for the communications and presentation layers), metric configuration and metadata.

Lifecycle management is invoked by the OSGi framework when loading and unloading a metric plugin, and is required to keep the databases clean. It follows a standard pattern of install, update and remove. Metric configuration and metadata is a straightforward keys-and-values kind of interface.

Metric plugins implement one or more metrics that are interested in one or more kinds of change in the open source projects under study. We use Java's reflection mechanism to dispatch requests; as a consequence the metric plugin API has a method `run(Object)` which takes an object describing the change and this is dispatched to the relevant measurement methods in each metric.

Figure 3 presents the steps required to calculate a simple line counting metric for an array of files. The updater component is notified externally that an update to the mirrored project assets has occurred; it then proceeds to incrementally process the asset metadata while recording the exact resources that have changed. It then passes



(a) The client requests the result of metric on a file

(b) The web service acquires a metric interface

(c) The web service asks the metric

(d) The metric retrieves the result

(e) The result is returned

Fig. 4. Results retrieval

the corresponding information to the metric activator (a). The metric activator creates a job for each changed asset and calls the scheduler to enqueue the jobs (b). When a thread becomes available, the scheduler runs the job. The job itself essentially calls the metric's `run` method with the appropriate argument (c), in our case an object encapsulating a file. With this reference available, the metric can query the FDS component to retrieve the file's contents from the original data source, in that case directly from the project's repository. Finally, the lines of the file are counted and the result is stored in the database (e).

The simplest result retrieval scenario is presented in Figure 4. The client asks the web service component for the measurement calculated by a metric on a specific file (a). As each metric can store results in arbitrary ways in the system's database, it is not possible to search for metric results using a generic data retrieval mechanism; instead each plug-in provides its own results retrieval function. Therefore, the web service must call the plug-in administrator to obtain a reference to the plug-in interface that implements the specific metric (b) and then query the plug-in itself for the result of the metric on the specific file (c). The plug-in code searches the database (d) for the result and returns it to the web service, which encapsulates it in a SOAP message and returns it to the client(e).

Measurement retrieval incurs a complication. Some measurements may not be done yet — for instance, when a large project is added it may take some time for all measurement to be completed or we may ignore measurements of "old" data until such time as someone expresses interest in them. We can distinguish situations in which response time is important; if the user interface in the presentation layer makes a request (through the communication layer and the core) for a specific measurement, it is important to give a quick response: the measurement value if we know it or otherwise an "I don't know yet" value (and then start calculating the

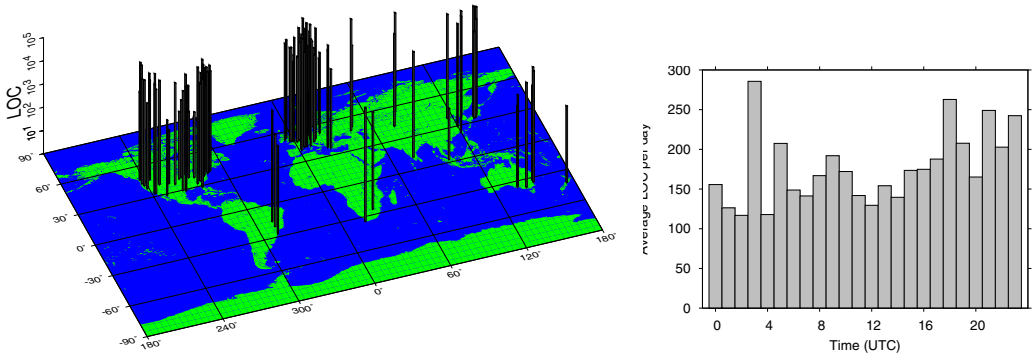


Fig. 5. Global and round-the-clock development in the FreeBSD system

measurement so as to return a better result next time). In other situations, response time is immaterial. Suppose we have a metric that calculates the ratio between two other measurements in the system (for instance, percentage of comments in a file, by dividing the number of comment lines by the total number of lines). Here, the calculation *must* have both values to proceed.

We introduce two methods for retrieving a measurement: a blocking one (which waits until the measurement is available) and a non-blocking one (which will return an "I don't know" value). A non-blocking compound measurement will use non-blocking retrieval for its constituents and a blocking compound measurement will use blocking retrieval. Non-blocking queries that return "I don't know" will start the measurement process so that future queries will receive a value. This approach neatly solves the problem of measurements-not-yet-available for the presentation layer and also the problem of compound metrics. The architecture of the Alitheia system can therefore be viewed in several ways: as a standard three-tier architecture; as a multi-level caching scheme to get data to metrics; as a storage provider for the metrics, and as a database of results that fills up in response to user queries.

4 Research on Open Source Software Quality

This section lists some motivating examples of how product and process metrics can provide insights into software quality.

4.1 Quality in Global Software Development

A pilot study preceding the development of the SQO-OSS infrastructure [28] used data assets from the FreeBSD operating system to examine the extent of global development and its effect on productivity and quality. Specifically, we used developer location data, the configuration management repository, and records from the issue database.

One often-claimed advantage of global software development is the ability to develop software round-the-clock in a continuous 24 hour cycle. In Figure 5 we can see that this goal is indeed realized in the FreeBSD project. Over a period of ten

years, FreeBSD developers committed on average 177 lines on every hour of each day; this number fluctuated between a minimum of 116 lines (at 02:00 UTC) and a maximum of 285 lines (at 03:00 UTC).

The study also examined how a large number of (geographically dispersed) committers might affect the quality of the produced code. If the software's quality deteriorates when software is globally developed, managers should appreciate this problem, and establish procedures for dealing with it. The quality of code is determined by many elements, and measuring it is far from trivial [32,22]; For the purpose of the study we chose to examine adherence to the FreeBSD code style guidelines [7] as a proxy for the overall code quality. We chose that metric because we could easily measure style adherence by formatting each source code file with the *indent* program configured according to the FreeBSD style guide, and calculate the percentage of lines that *indent* would change (the size of a minimal set of differences between the actual file and the formatted one). Furthermore, by having CVS generate a listing of the source code file with every line annotated with the name of the author who last modified it, we could count the number of developers who had worked on the file.

Armed with those two measurements, we used Pearson's product-moment method to examine the correlation between the two. The correlation coefficient for the 11,040 pairs of measurements was a miserly 0.05 in a 95% confidence interval between 0.03 and 0.07. We therefore saw that in the case of FreeBSD, the involvement of geographically dispersed programmers (a process attribute) in the development of code did not affect the quality of the produced code (a product attribute).

Finally, we examined whether the global development of a file by various developers was associated with an increased number of problem reports filed for it. Such a correlation could indicate that global development in the FreeBSD project leads to an increased number of bugs in the code, due, for example, to communication problems between the various developers. Although problem reports are kept in a database different from that of the FreeBSD configuration management system, rectified problems are typically marked in a CVS commit message by a reference to the corresponding problem report (PR). Because serious problem reports are by definition sooner or later rectified, we could establish a measure of the density of problem reports in a file by dividing the number of commit messages tagged with a PR number with the total number of the file's commits. We could then examine the correlation of that ratio with the number of different developers that had committed code to the corresponding file.

We collected data for 33,392 source code files, 457,481 commit messages, and 12,505 PRs. On average, each file was associated with 13.7 commits, 0.37 PRs, and 4.2 different developers. A two sided Pearson's product-moment correlation test between the PR density and the number of committers gave an insignificant correlation between the two values (0.07) in a 95% confidence interval between 0.06 and 0.08. Therefore, the data from the FreeBSD project did not support the hypothesis that global software development is associated with a higher bug density

in the code produced.

4.2 Mean Developer Engagement

The principles behind the agile development methods and common practice within the Open Source community are vastly different. In recent years there has been a rise of interest in these, in order to detect and inform on areas of compatible shared practices. In [1] we argue that it is possible to quantify the level of agility displayed by Open Source projects. An indicator of agility, the Mean Developer Engagement (MDE) metric is introduced and tested through the analysis of public project data. Projects sampled from two repositories (KDE and SourceForge) are studied and a null hypothesis is formulated: projects from the two samples display a similar level of MDE.

As developers are a limited resource within the Free Software community it is important that Free Software projects engage their developer resource in order to maintain their interest. To this end the MDE metric for measuring engagement is defined as "the ability, on average, over the lifetime of a Free Software project, for that project to make use of its developer resources." Mathematically this can be described as:

$$\bar{de} = \frac{\sum_{i=1}^{n} \left(\frac{\text{dev}(active)}{\text{dev}(total)} \right)_i}{n} \tag{1}$$

Where:

- dev(*active*) is the number of (distinct) developers active in time period i.
- dev(*total*) is the total number of developers involved with the project in the periods $0 \ldots i$.
- n is the number of time periods over which the project has been evaluated. For this research these were periods of a week.

The initial failing of this approach is that dev(*total*) is hard to define. As a first attempt this was simply taken to be the number of accounts within the project Subversion repository. This, however, is a naïve approach, because account details remain within a Subversion repository even after the developer has left the project. To make this measurement more accurate we introduced a developer "grace period". This is a period of developer inactivity where we still consider the developer part of dev(*total*). The longer a developer has been involved with a project, the longer we allow their grace period to grow.

An example plot of MDE for the history of the KDE projects is provided in Fig. 6. This plot clearly shows some features common to all Open Software projects:

- MDE, mathematically, must start at 1. That is, at least one developer was active within the first week of the project and, at the time, was the only developer in the project. Put simply, the projet is founded.
- The MDE at the begining of the project shows a fluctuation. This is caused by changes in dev(*active*) whilst both dev(*total*) and n are low.

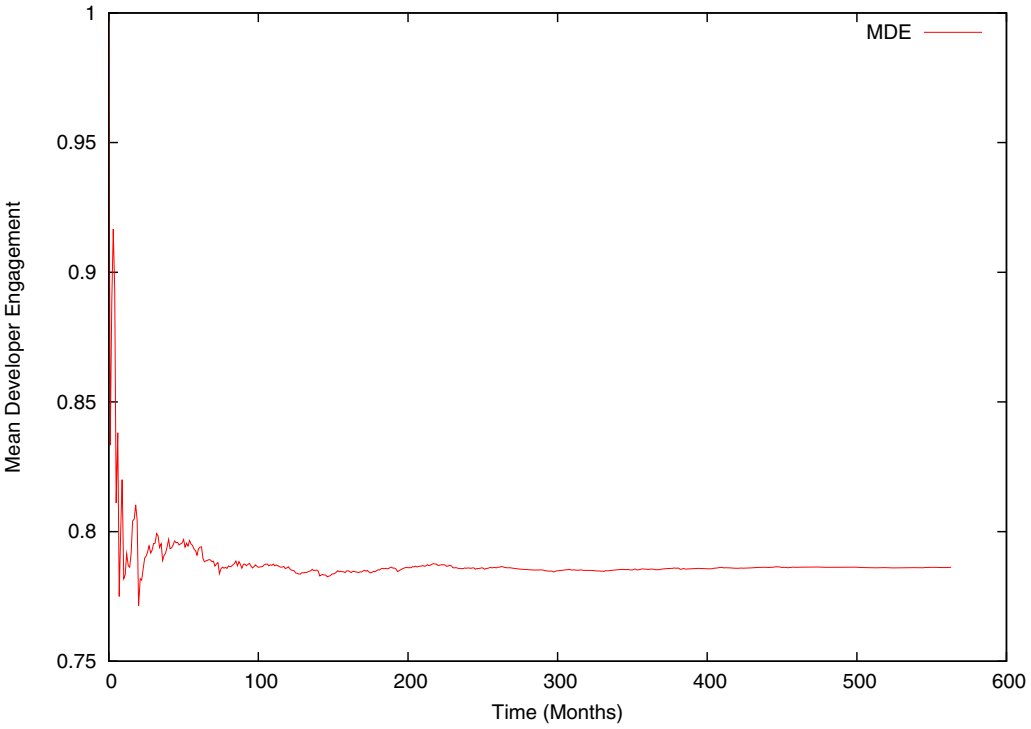


Fig. 6. MDE for the KDE Project

- The period of fluctutation is followed by a period of greater stability.

A unique element to Fig. 6 is that the stable phase of the project displays no particular trend (up or downward). Instead the project has successfully maintained a near 80% level of activity for close to a decade.

We applied MDE to the entire history of 40 Open Source projects, 20 randomly selected from within KDE and 20 from within SourceForge.net. We then used the Wilcoxon test to show that, over their lifetime, projects from SourceForge show a significantly higher MDE than KDE projects. This however held a strong correlation with a n value of only 1 week. To counteract this, we finally produced an "effort" score. To do this, we first calculated the average MDE over the lifetime of the project by taking the MDE value for each week of the project's lifetime and then making a simple average. The effort score was simply calculated by multiplying this average MDE score by the maximum value of n, the length of the project in weeks. By comparing the new effort values we were able to reapply the Wilcoxon test to find a result of $W = 374, p \leq 2.405e - 06$. This allowed us to state with 95% certainty that a randomly selected KDE will show greater engagement of its developers over time than a randomly selected SourceForge project and therefore the codebase will encapsulate greater developer effort.

Metric Id	Metric Name	SQO-OSS v0.8
NOPA	Number of Public Attributes	•
NOC	Number of Children	•
DIT	Depth of Inheritance Tree	
AC	Afferent Couplings	
NPM	Number of Public Methods	
RFC	Response for a Class	
LOC	Lines of Code	•
COM	Lines of Comments	•
LCOM	Lack of Cohesion in Methods	
NOPRM	Number of Protected Methods	•
NOCL	Number of Classes	•
CBO	Coupling between Object Classes	
WMC	Weighted Methods per Class	•

Table 1
Supported metrics by CLMT

4.3 Cross-Language Metric Tool

The Cross-Language Metric Tool (CLMT) calculates software complexity metrics in a variety of programming languages. CLMT implements this by transforming each programming language to an Intermediate XML Representation (IXR).

CLMT accepts as input an XML file that describes specific metric calculation tasks. The source files are parsed and IXR files are generated. The metrics are calculated through a series of queries on the IXR. The results are presented as textual output or as XML structured documents.

Table 1 lists the metrics that will be supported in the first CLMT release. In the third column we show the metrics that are already supported in the current version. Although CLMT now works with Java programming language, support for C and C++ is planned for the next release; in fact, support for different programming languages was one of the main design requirements.

CLMT leads an independent existence from SQO-OSS as a stand-alone application, but has also been integrated with the latter, although the integration was not straightforward. In Figure 7 we depict the CLMT architecture as a SQO-OSS plug-in.

4.4 Unneeded Header File Include Directives

A number of widely used programming languages use lexically included files as a way to share and encapsulate declarations, definitions, code, and data. As the code evolves files included in a compilation unit are often no longer required, yet locating and removing them is a haphazard operation, which is therefore neglected. Needlessly included files are detrimental to the quality of a project, because they contribute to namespace pollution, they introduce spurious dependencies, and they increase compilation time. The difficulty of reasoning about included files stems primarily from the fact that the definition and use of macros complicates the notions of scope and of identifier boundaries. By defining four successively refined identifier equivalence classes we can accurately derive dependencies between identifiers [26]. A mapping of those dependencies on a relationship graph between included files can

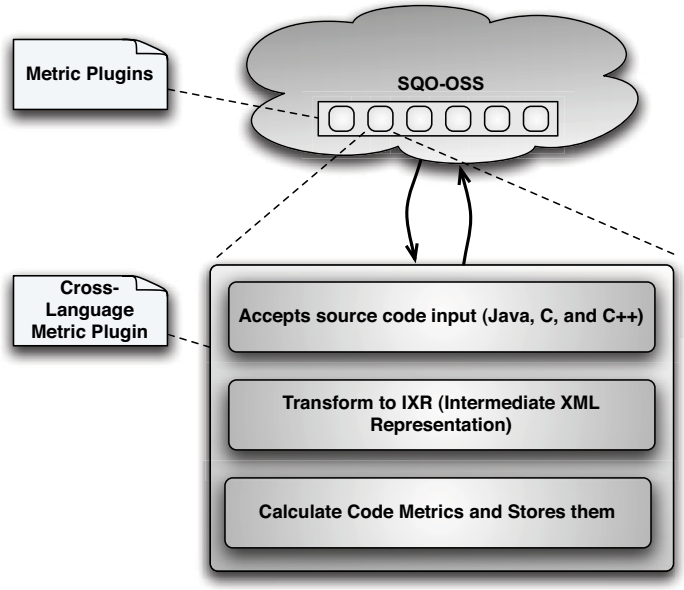


Fig. 7. The CLMT architecture

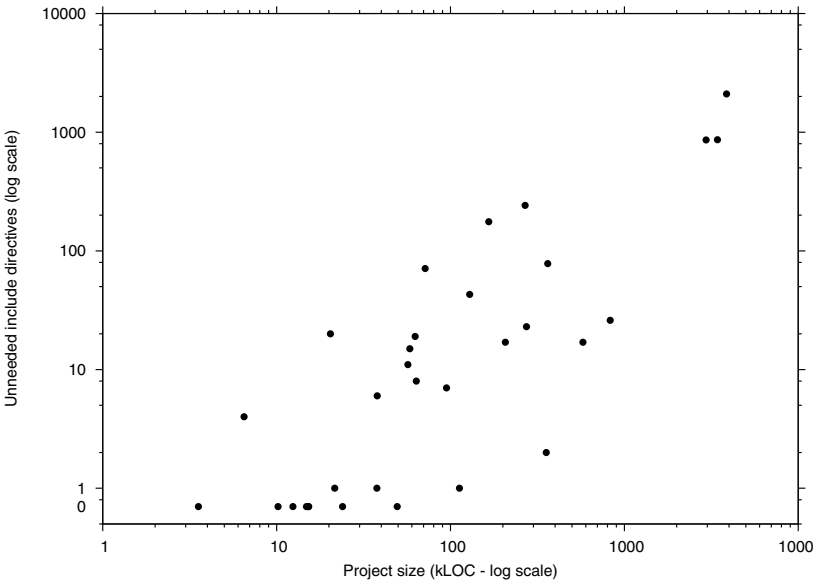


Fig. 8. Unneeded include directives in projects of various sizes.

then be used to determine included files that are not required in a given compilation unit [29]. Specifically, a header file is required only if it

- contains a definition for an identifier;
- includes another file that is required; or
- provides code or data to the compilation unit that includes it.

We tested our approach on 32 medium and large–sized open-source projects. These were: the Apache httpd 1.3.27, Lucent's awk as of Mar 14th, 2003, bash 3.1, CVS 1.11.22, Emacs 22.1, the kernel of FreeBSD HEAD branch as of September 9th, 2006 LINT configuration processed for the i386, AMD64, and SPARC64 architectures, gdb 6.7, Ghostscript 7.05, gnuplot 4.2.2, AT&T GraphViz 2.16, the default configuration of the Linux kernel 2.6.18.8-0.5 processed for the x86-64 (AMD64) architecture, the kernel of OpenSolaris as of August 8th, 2007 configured for the Sun4v Sun4u and SPARC architectures, the Microsoft Windows Research Kernel 1.2 processed for the i386 and AMD64 architectures, Perl 5.8.8, PostgreSQL 8.2.5, Xen 3.1.0, and the versions of the programs bind, ed, lex, mail, make, ntpd, nvi, pax, pppd, routed, sendmail, tcpdump, tcsh, window, xlint, and zsh distributed with FreeBSD 6.2. The FreeBSD programs were processed under FreeBSD 6.2 running on an i386 processor architecture, while the rest, where not specified, were configured under openSUSE Linux 10.2 running on an AMD64 processor architecture. For expediency, we selected the projects by looking for representative, widely-used, large-scale systems that were written in C and could be compiled standalone. The processed source code size was 14.2 million lines of code.

A summary of the results appears in Figure 8. As we can see, unneeded header files are rarely a problem for projects smaller than 20 KLOC, but become a significant one as the project's size increases. (The chart's abscissa also includes a notional value of zero where projects without include directive problems are indicated.)

4.5 A Metric for Developer Contributions

In software engineering, contribution assessment entails the measurement of the contribution of a person in terms of lines of code or function points towards the development of a software project. In the recent years however, the shift to more agile development practices and the proliferation of software and project management tools has reduced the estimation capacity of classic software estimation models. A software developer today is not only required to write code, but also to communicate with colleagues effectively and to use a variety of tools that produce and modify code with minimal input from his side.

In [9] we present a model that exploits the availability of publicly accessible software repositories to extract process data and combines them in a single contribution factor. Table 2 presents an overview of actions on project assets that our model evaluates. The number of actions for each action type is calculated per developer, while weights are applied to each action, depending on how often this action appears across an array of projects. The extracted contribution factor is then combined with the developer's total lines of code to extract the developer's contribution.

4.6 The Effects of Refactoring on Software Quality

Refactoring is considered as one of the most important means of transforming a piece of software in order to improve its quality. Its aim is to decrease the complexity of a system at design and source code level, allowing it to evolve further in a low-cost

Asset	Action	Type
Code and Documentation Repository	Add lines of code of good/bad quality	P/N
	Commit new source file or directory	P
	Commit code that generates/closes a bug	N/P
	Add/Change code documentation	P
	Commit fixes to code style	P
	Commit more than X files in a single commit	N
	Commit documentation files	P
	Commit translation files	P
	Commit binary files	N
	Commit with empty commit comment	N
	Commit comment that awards a pointy hat	P
	Commit comment that includes a bug report num	P
Mailing lists - Forums	First reply to thread	P
	Start a new thread	P
	Participate in a flamewar	N
	Close a lingering thread	P
Bug Database	Close a bug	P
	Report a confirmed/invalid bug	P/N
	Close a bug that is then reopened	N
	Comment on a bug report	P
Wiki	Start a new wiki page	P
	Update a wiki page	P
	Link a wiki page from documentation/mail file	P
IRC	Frequent participation to IRC	P
	Prompt replies to directed questions	P

Table 2
Project resources and actions that can be performed on them. The Type column denotes whether an action has positive (P) or negative (N) impact.

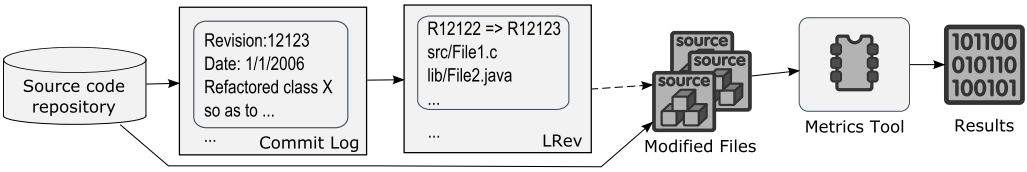


Fig. 9. The process used to evaluate the effect of refactorings

manner by ensuring the developers' productivity and leaving less room for design errors[18]. Here we are interested in the effect of refactorings on the quality of well known open source projects, as presented in our study reported in [34].

Most of the studies examining the relation between software quality and metrics, like [35,14,33], or refactoring and software quality, do not correlate the evolution of a system with changes in metrics measurements. We tried to show how refactoring has affected metrics in popular open source software projects. Various established software quality metrics were measured before and after the application of refactoring transformations. The source control system history was used as a source of information to detect the refactorings performed between consequent revisions.

A surprising finding of this study has been that refactoring has an adverse effect on the values of software quality metrics on a sample of 4 OSS projects. Specifically it seems that refactoring caused a non trivial increase in metrics such as LCOM

Dataset	size	k in/out	r^2 in/out
J2SE SDK	13,055	2.09/3.12	.99/.86
Eclipse	22,001	2.02/3.15	.99/.87
OpenOffice	3,019	1.93/2.87	.99/.94
BEA WebLogic	80,095	2.01/3.52	.99/.86
CPAN packages	27,895	1.93/3.70	.98/.95
Linux libraries	4,047	1.68/2.56	.92/.62
FreeBSD libraries	2,682	1.68/2.56	.91/.58
MS-Windows binaries	1,355	1.66/3.14	.98/.76
FreeBSD ports, libraries deps	5,104	1.75/2.97	.94/.76
FreeBSD ports, build deps	8,494	1.82/3.50	.99/.98
FreeBSD ports, runtime deps	7,816	1.96/3.18	.99/.99
TEX	1,364	2.00/2.84	.91/.85
META-FONT	1,189	1.94/2.85	.96/.85
Ruby	603	2.62/3.14	.97/.95
The errors of TEX	1,229	3.36	.94
Linux system calls (242)	3,908	1.40	.89
Linux C libraries functions (135)	3,908	1.37	.84
FreeBSD system calls (295)	3,103	1.59	.81
FreeBSD C libraries functions (135)	3,103	1.22	.80

Table 3
Software Power Laws.

(Lack of Cohesion in Methods, expresses the similarity of methods), Ca (Afferent Coupling, the number of other packages depending upon a class) and RFC (Response for a Class, the sum of the number of methods of the class itself and all other methods it calls), indicating that it caused classes to become less coherent as more responsibilities are assigned to them. The same principles seem to apply in procedural systems as well, in which case the effect is captured as an increase in complexity metrics. Since it is a common conjecture that the metrics used can actually indicate a system's quality, these results suggest that either the refactoring process does not always improve the quality of a system in a measurable way or that developers still have not managed to use refactoring effectively as a means to improve software quality. In other words, these results may indicate that either refactoring was not used in a way that improves the quality of the studied projects or that software quality metrics are not the best method to measure the quality improvements introduced by refactoring.

4.7 Power Laws in Software

The notion of power laws as a descriptive device has been around for more than a century [20]. During this period power laws have cropped up in different guises in various contexts. Mathematically, a power law is a probability distribution function in which the probability that a random variable takes a value is proportional to a negative power of that value:

$$P(X = x) \propto cx^{-k} \quad \text{where } c > 0,\ k > 0 \tag{2}$$

The availability of large open-source software systems allowed us to study the existence of scale-free networks of their modules [16]. We chose modules of varying size and functionality, ranging from simple Java classes to systems using self-contained libraries written in C, Perl, and Ruby. For our purposes, the links connecting the modules are given by their dependencies. For two modules A and B we add a directed link from B to A when B depends on A. This produces a directed graph. We explore the structure of both the incoming links and the outgoing links.

Note that measuring fan-in and fan-out is not new, and has been used as a measure for procedural complexity [11]. Here we are not interested in measuring complexity, but in seeing whether incoming and outgoing links *in different levels of abstraction* show similar patterns. Such patterns could then be related to various quality metrics.

A summary of our findings is shown in Table 3. In each row we list the number of nodes, the exponent for the incoming links and outgoing links, where applicable, and the corresponding correlation coefficient. The long, fat tails observed in our data impact on several aspects of software engineering, such as quality, design, reuse, and optimization. Based on our results, we propose taking into account the power laws present in software to focus development efforts and save resources in quality assurance tasks.

Even though, as software developers, we may not be able to locate troublespots in a system, we have a measure of the impact of our efforts. Selecting modules at random, we may expect that around a percent of the dependencies will not lead to bugs propagated from bugs in the selected modules. If, however, we focus on the top (in terms of dependent modules) a percent of the modules in a system, we may avoid the propagation of errors to up to a^θ other dependent modules, where $\theta = 1 - \frac{1}{k-1}$ (for details see [16])—a significant improvement.

For instance, the success and failure of beta-testing can be illuminated if we consider the scale-free distribution of bugs; beta-testers will discover quickly the small number of defects that make up a large proportion of those that can be found; at the same time, there will always be other effects, with a much lower probability to be found during testing, that will continue to torment unlucky users during production. However, despite the best of efforts, a system may still fail. Recovery-Oriented Computing accepts this as a fact of life and demands that systems appropriate for rapid recovery should be identified at various levels of abstraction [2]. This suggests that hub modules could be suitable candidates.

4.8 A Quality Model for Open Source Software

In the context of the SQO-OSS project we defined a model for software quality evaluation, based on software models [15] that define and measure software quality. This particular model aims at capturing the particularities arising from the special nature of open source software development process. Moreover, it focuses both on source code and the community around a project.

The model is presented in [23]. The model construction process followed a GQM [25] approach. The outcome was a hierarchical tree view of the quality at-

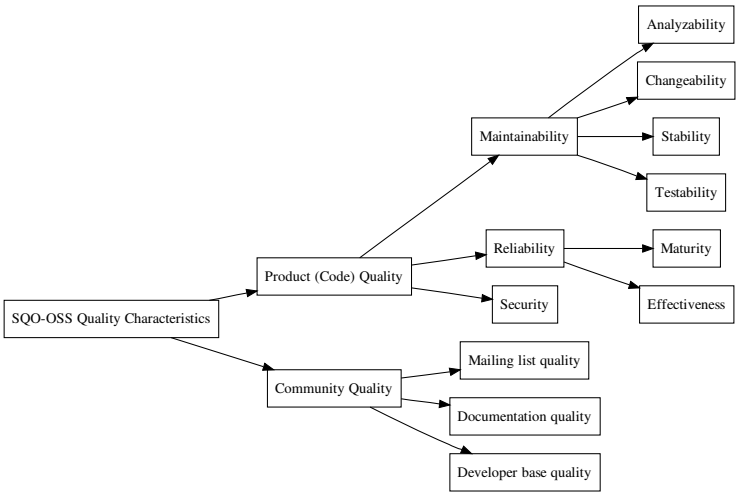


Fig. 10. The SQO-OSS quality model

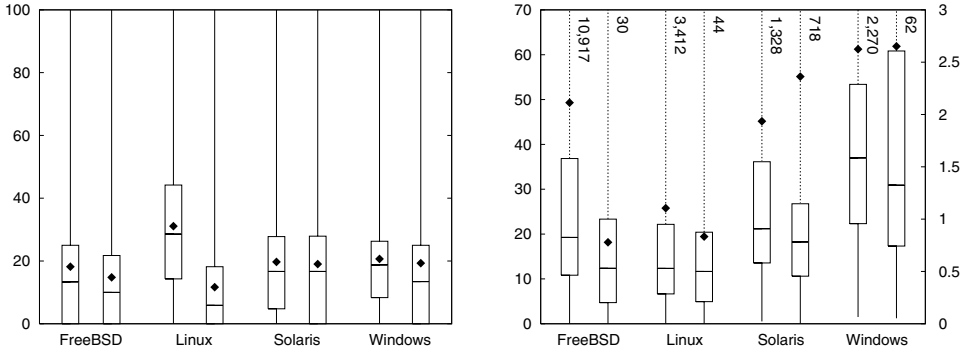


Fig. 11. Common coupling at file and global scope (left); comment density in C and header files (right).

tributes, shown in Figure 10. The leaves of the tree are further analyzed into metrics measured by the system and used for evaluation of selected criteria. The metrics are presented in Table 4. In order to combine all the measurements in one single view, that is to aggregate the measurements, we used the profile based evaluation process described in detail in [21]. Profile based aggregation allows us to categorize software quality into four categories: *Excellent, Good, Fair* and *Poor*. For these categories, we constructed corresponding profiles, with certain measurement values indicated by the existing literature. For example, the ideal candidate for the *Excellent Analyzability* quality attribute should have a McCabe Cyclomatic number equal to 4, Average function's number of statements equal to 10, Comments frequency equal to 0.5 and an Avarage size of statements equal to 2.

Attribute	Metric
Analyzability	Cyclomatic number
	Number of statements
	Comments frequency
	Average size of statements
	Weighted methods per class (WMC)
	Number of base classes
	Class comments frequency
Changeability	Average size of statements
	Vocabulary frequency
	Number of unconditional jumps
	Number of nested levels
	Coupling between objects (CBO)
	Lack of cohesion (LCOM)
	Depth of inheritance tree (DIT)
Stability	Number of unconditional jumps
	Number of entry nodes
	Number of exit nodes
	Directly called components
	Number of children (NOC)
	Coupling between objects (CBO)
	Depth of inheritance tree (DIT)
Testability	Number of exits of conditional structs
	Cyclomatic number
	Number of nested levels
	Number of unconditional jumps
	Response for a class (RFC)
	Average cyclomatic complexity per method
	Number of children (NOC)
Maturity	Number of open critical bugs in the last 6 months
	Number of open bugs in the last six months
Effectiveness	Number of critical bugs fixed in the last 6 months
	Number of bugs fixed in the last 6 months
Security	Null dereferences
	Undefined values
Mailing list	Number of unique subscribers
	Number of messages in user/support list per month
	Number of messages in developers list per month
	Average thread depth
Documentation	Available documentation documents
	Update frequency
Developer base	Rate of developer intake
	Rate of developer turnover
	Growth in active developers
	Quality of individual developers

Table 4
Metrics used by the SQO-OSS Quality Model

4.9 A Comparison of Four Operating System Kernels

In another study [30] we looked at quality differences between software developed as a proprietary product and software developed in an open-source fashion. Specifically, the FreeBSD, GNU/Linux, Solaris, and Windows (WRK) operating systems have

Metric		FreeBSD	Linux	Solaris	WRK
A. Overview					
Version		2006-09-18	2.6.18.8-0.5	2007-08-28	1.2
Lines (thousands)		2,599	4,150	3,000	829
Comments (thousands)		232	377	299	190
Statements (thousands)		948	1,772	1,042	192
Source files		4,479	8,372	3,851	653
Linked modules		1,224	1,563	561	3
C functions		38,371	86,245	39,966	4,820
Macro definitions		727,410	703,940	136,953	31,908
B. File Organization					
Files per directory	↘	6.8	20.4	8.9	15.9
Header files per C source file	≈ 1	1.05	1.96	1.09	1.92
Average structure complexity in files	↘	$2.2\ 10^{14}$	$1.3\ 10^{13}$	$5.4\ 10^{12}$	$2.6\ 10^{13}$
C. Code Structure					
% global functions	↘	36.7	21.2	45.9	99.8
% strictly structured functions	↗	27.1	68.4	65.8	72.1
% labeled statements	↘	0.64	0.93	0.44	0.28
Average # function parameters	↘	2.08	1.97	2.20	2.13
Average depth of maximum nesting	↘	0.86	0.88	1.06	1.16
Tokens per statement	↘	9.14	9.07	9.19	8.44
% tokens in replicated code	↘	4.68	4.60	3.00	3.81
Average function structure complexity	↘	$7.1\ 10^{4}$	$1.3\ 10^{8}$	$3.0\ 10^{6}$	$6.6\ 10^{5}$
D. Code Style					
% style conforming lines	↗	77.27	77.96	84.32	33.30
% style conforming typedef identifiers	↗	57.1	59.2	86.9	100.0
% style conforming aggregate tags	↗	0.0	0.0	20.7	98.2
Characters per line	↘	30.8	29.4	27.2	28.6
% numeric constants in operands	↘	10.6	13.3	7.7	7.7
% unsafe function-like macros	↘	3.99	4.44	9.79	4.04
% misspelled comment words	↘	33.0	31.5	46.4	10.1
% unique misspelled comment words	↘	6.33	6.16	5.76	3.23
E. Preprocessing					
% preprocessor directives in header files	↘	22.4	21.9	21.6	10.8
% non-#include directives in C files	↘	2.2	1.9	1.2	1.7
% preprocessor directives in functions	↘	1.56	0.85	0.75	1.07
% preprocessor conditionals in functions	↘	0.68	0.38	0.34	0.48
% function-like macros in functions	↘	26	20	25	64
% macros in unique identifiers	↘	66	50	24	25
% macros in identifiers	↘	32.5	26.7	22.0	27.1
F. Data Organization					
% global scope variable declarations	↘	0.36	0.19	1.02	1.86
% global scope variable operands	↘	3.3	0.5	1.3	2.3
% identifiers with wrongly global scope	↘	0.28	0.17	1.51	3.53
% variable declarations with file scope	↘	2.4	4.0	4.5	6.4
% variable operands with file scope	↘	10.0	6.1	12.7	16.7
Variables per typedef or aggregate	↘	15.13	25.90	15.49	7.70
Data elements per aggregate or `enum`	↘	8.5	10.0	8.6	7.3

Metric interpretation: ↘ means lower is better; ↗ means higher is better.

Table 5
Key scalar metrics for four operating system kernels

kernels that provide comparable facilities, but their code bases share almost no common parts, while their development processes vary dramatically. We analyzed the source code of the four systems by collecting metrics in the areas of file organization, code structure, code style, the use of the C preprocessor, and data organization (see Table 5 and Figure 11). The aggregate results indicated that across various areas and many different metrics, four systems developed using wildly different processes scored comparably. This allowed us to posit that the structure and internal quality attributes of a working, non-trivial software artifact will represent first and foremost the engineering requirements of its construction, with the influence of process being marginal, if any.

5 Conclusions

By combining both product and process metrics we are able to answer novel issues in software development, with particular emphasis on quality aspects. In SQO-OSS we take that view in earnest, and we have designed and implemented a platform that allows both kinds of data to be captured and analysed in an efficient way.

It is important to note that SQO-OSS is not (another) metrics evaluation system. It is a platform on which metrics can be developed, plugged it, and run, on projects of any size. Our plans include extending and maintaining SQO-OSS so as to function as a digital repository for Open Source software research.

As shown in Section 4, we have already tackled a number of interesting research questions based on quantitative measurements of quality attributes of Open Source projects. The availability of an open platform for supporting this sort of inquiry will enable us to pursue further research questions; we also hope that other researchers will wish to take advantage of our infrastructure by working on new metrics and evaluating them with large scale measurements on SQO-OSS.

References

[1] Adams, P. J., A. Capiluppi and A. de Groot, *Detecting agility of open source projects through developer engagement*, in: *OSS2008: Proceedings of the Fourth International Conference on Open Source Systems*, 2008.

[2] Candea, G., A. B. Brown, A. Fox and D. Patterson, *Recovery-oriented computing: Building multitier dependability*, IEEE Computer **37** (2004), pp. 60–67.

[3] Coleman, D., D. Ash, B. Lowther and P. W. Oman, *Using metrics to evaluate software system maintainability*, Computer **27** (1994), pp. 44–49.

[4] Cusumano, M. A. and R. W. Selby, "Microsoft Secrets," The Free Press, New York, 1995.

[5] Debenest, P., E. F. Fukushima, Y. Tojo and S. Hirose, *A new approach to humanitarian demining. part 1: Mobile platform for operation on unstructured terrain*, Autonomous Robots **18** (2005), pp. 303–321.

[6] Forrester, J. E. and B. P. Miller, *An empirical study of the robustness of Windows NT applications using random testing*, in: *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium* (2000), pp. 59–68.

[7] The FreeBSD Project, "Style—Kernel Source File Style Guide," (1995), freeBSD Kernel Developer's Manual: style(9). Available online http://www.freebsd.org/docs.html (January 2006).

[8] Ghosh, R. A., *Study on the: Economic impact of open source software on innovation and the competitiveness of the information and communication technologies (ICT) sector in the EU*, Available online http://ec.europa.eu/enterprise/ict/policy/doc/2006-11-20-flossimpact.pdf. (2006), prepared by MERIT for the European Commission under the contract ENTR/04/112.

[9] Gousios, G., E. Kalliamvakou and D. Spinellis, *Measuring developer contribution from software repository data*, in: A. E. Hassan, M. Lanza and M. W. Godfrey, editors, *MSR '08: Mining Software Repositories* (2008), pp. 129–132.

[10] Hassan, A. E., R. C. Holt and A. Mockus, *Report on MSR 2004: International workshop on mining software repositories*, SIGSOFT Software Engineering Notes **30** (2005), p. 4.

[11] Henry, S. and D. Kafura, *Software structure metrics based on information flow*, IEEE Transactions on Software Engineering **7** (1981), pp. 510–518.

[12] International Organization for Standardization, *Software Engineering—Product Quality—Part 1: Quality Model* (2001), iSO/IEC 9126-1:2001.

[13] Jørgensen, N., *Putting it all in the trunk: Incremental software development in the FreeBSD open source project*, Information Systems Journal **11** (2001), pp. 321–336.

[14] Kan, S. H., *Metrics and models in software quality engineering (2nd edition)* (2002).
URL http://www.amazon.co.uk/exec/obidos/ASIN/0201729156/citeulike-21

[15] Kanellopoulos, Y., I. Heitlager, C. Tjortjis and J. Visser, *Interpretation of source code clusters in terms of the ISO/IEC-9126 maintainability characteristics*, in: K. Kontogiannis, C. Tjorjis and A. Winter, editors, *12th European Conference on Software Maintenance and Reengineering* (2008), pp. 63–72.

[16] Louridas, P., D. Spinellis and V. Vlachos, *Power laws in software*, ACM Transactions on Software Engineering and Methodology (2008), to appear.

[17] Matthies, L., M. Maimone, A. Johnson, Y. Cheng, R. Willson, C. Villalpando, S. Goldberg, A. Huertas, A. Stein and A. Angelova, *Computer vision on Mars*, International Journal of Computer Vision **75** (2007), pp. 67–92.

[18] Mens, T. and T. Tourwé, *A Survey of Software Refactoring*, IEEE Transactions on Software Engineering **30** (2004), pp. 126–139.

[19] Miller, B. P., L. Fredriksen and B. So, *An empirical study of the reliability of UNIX utilities*, Communications of the ACM **33** (1990), pp. 32–44.

[20] Mitzenmacher, M., *A brief history of generative models for power law and lognormal distributions*, Internet Mathematics **1** (2004), pp. 226–251.

[21] Morisio, M., I. Stamelos and A. Tsoukias, *Software product and process assessment through profile-based evaluation*, International Journal of Software Engineering and Knowledge Engineering **13** (2003), pp. 495–512.

[22] Payne, C., *On the security of open source software*, Information Systems Journal **12** (2002), pp. 61–78.

[23] Samoladas, I., G. Gousios, D. Spinellis and I. Stamelos, *The sqo-oss quality model: measurement based open source software evaluation*, in: *OSS '08: Proceedings of the International Conference on Open Source Systems 2008* (2008), to appear.

[24] Samoladas, I., I. Stamelos, L. Angelis and A. Oikonomou, *Open source software development should strive for even greater code maintainability*, Communications of the ACM **47** (2004), pp. 83–87.

[25] Solingen, R. V., *The goal/question/metric approach*, Encyclopedia of Software Engineering **2** (2002), pp. 578–583.

[26] Spinellis, D., *Global analysis and transformations in preprocessed languages*, IEEE Transactions on Software Engineering **29** (2003), pp. 1019–1030.

[27] Spinellis, D., "Code Quality: The Open Source Perspective," Addison-Wesley, Boston, MA, 2006.

[28] Spinellis, D., *Global software development in the FreeBSD project*, in: P. Kruchten, Y. Hsieh, E. MacGregor, D. Moitra and W. Strigel, editors, *International Workshop on Global Software Development for the Practitioner* (2006), pp. 73–79.

[29] Spinellis, D., *Optimizing header file include directives*, Journal of Software Maintenance and Evolution: Research and Practice (2008), .To appear.

[30] Spinellis, D., *A tale of four kernels*, in: W. Schäfer, M. B. Dwyer and V. Gruhn, editors, *ICSE '08: Proceedings of the 30th International Conference on Software Engineering* (2008), pp. 381–390.

[31] Spinellis, D. and C. Szyperski, *How is open source affecting software development?*, IEEE Software **21** (2004), pp. 28–33.

[32] Stamelos, I., L. Angelis, A. Oikonomou and G. L. Bleris, *Code quality analysis in open source software development*, Information Systems Journal **12** (2002), pp. 43–60.

[33] Stamelos, I., L. Angelis, A. Oikonomou and G. L. Bleris, *Code quality analysis in open source software development.*, Information Systems Journal **12** (2002), pp. 43–60.

[34] Stroggylos, K. and D. Spinellis, *Refactoring: Does it improve software quality?*, in: B. Boehm, S. Chulani, J. Verner and B. Wong, editors, *5th International Workshop on Software Quality* (2007), pp. 1–6.

[35] Subramanyam, R. and M. S. Krishnan, *Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects*, IEEE Transactions on Software Engineering **29** (2003), pp. 297–310.