Measuring and tracking quality factors in Free and Open Source Software projects

Fabian Fagerholm

Helsinki October 9, 2007

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty

Faculty of Science

Laitos — Institution — Department

Department of Computer Science

Tekijä — Författare — Author

Fabian Fagerholm

Työn nimi — Arbetets titel — Title

Measuring and tracking quality factors in Free and Open Source Software projects

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level

M. Sc. Thesis

Aika — Datum — Month and year

October 9, 2007

Sivumäärä — Sidoantal — Number of pages

77 pages

Tiivistelmä — Referat — Abstract

Free and Open Source Software (FOSS) has gained increased interest in the computer software industry, but assessing its quality remains a challenge. FOSS development is frequently carried out by globally distributed development teams, and all stages of development are publicly visible. Several product and process-level quality factors can be measured using the public data.

This thesis presents a theoretical background for software quality and metrics and their application in a FOSS environment. Information available from FOSS projects in three information spaces are presented, and a quality model suitable for use in a FOSS context is constructed. The model includes both process and product quality metrics, and takes into account the tools and working methods commonly used in FOSS projects.

A subset of the constructed quality model is applied to three FOSS projects, highlighting both theoretical and practical concerns in implementing automatic metric collection and analysis. The experiment shows that useful quality information can be extracted from the vast amount of data available. In particular, projects vary in their growth rate, complexity, modularity and team structure.

ACM Computing Classification System (CCS):

Categories and subject descriptors:

D.2.8 [Software Engineering]: Metrics;

D.2.9 [Software Engineering]: Management—Life cycle, Programming teams, Software configuration management, Software process models (e.g., CMM, ISO, PSP), Software quality assurance (SQA);

H.4.2 [Information systems applications]: Types of systems—Decision support (e.g., MIS);

K.6.3 [Management of computing and information systems]: Software management

General terms:

Human Factors, Management, Measurement

Avainsanat — Nyckelord — Keywords

free software, open source software, FOSS, software quality, software metrics

Säilytyspaikka — Förvaringsställe — Where deposited

Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Quality as a measurable quantity **2**

2.1 The intuitive view of quality 3

2.2 Translating intuition into a quality system 6

2.3 Metrics as the building blocks of quality measurement 9

2.3.1 Analysis and design metrics 11

2.3.2 Code metrics 12

2.3.3 Object-oriented metrics 13

2.3.4 Maintenance metrics 16

2.3.5 General metrics problems 17

3 Quality in the context of Free and Open Source Software **19**

3.1 The FOSS mind set: catalyst or obstacle? 20

3.1.1 How FOSS compares to process models 22

3.1.2 Process maturity and success 24

3.1.3 Openness and reliability 26

3.2 Previous work and case studies 26

3.2.1 Growth, evolution and structural change 27

3.2.2 Mining public repositories 28

3.2.3 Empirical tests of statistical quality models 31

3.2.4 Object-oriented metrics for FOSS 33

3.2.5 FOSS quality assessment model 35

4 Quality model for Free and Open Source Software projects **36**

4.1 Preliminary description and project hypotheses 36

4.2 Data sources . . . 38
4.2.1 Discussion space sources . . . 39
4.2.2 Documentation space sources . . . 39
4.2.3 Implementation space sources . . . 41
4.3 Data model . . . 42
4.4 Metric taxonomy . . . 43

5 Experiment: Applying quality analysis to real Free and Open Source Software projects 45
5.1 Project selection . . . 46
5.2 Data identification, acquisition and cleaning . . . 46
5.3 Description of metrics calculation . . . 50

6 Experiment: Results 51
6.1 Linux . . . 53
6.1.1 Process quality . . . 53
6.1.2 Product quality . . . 54
6.2 Blender . . . 58
6.2.1 Process quality . . . 58
6.2.2 Product quality . . . 60
6.2.3 Object-oriented features . . . 62
6.3 The GIMP . . . 63
6.3.1 Process quality . . . 64
6.3.2 Product quality . . . 65
6.4 Summary . . . 67

7 Conclusions 69

References 70

List of Figures

1 Software Quality Characteristics Tree 7
2 IEEE software quality metrics framework 8
3 Layered organisational structure of an idealised FOSS project 25
4 Fact extraction process of Columbus framework 34
5 FOSS project data model 43
6 FOSS project quality model 44
7 Commits per author: Linux and Linux-historical 53
8 Posts per poster and bugs per submitter: Linux 55
9 SLOC evolution: Linux and Linux-historical 55
10 Subsystem growth: Linux and Linux-historical 56
11 Commit and post frequency: Linux and Linux-historical 56
12 Bug frequency and arrival rate per commit rate: Linux 57
13 SLOC per NOM and CYC per NOM: Linux 57
14 Fan-in and fan-out per SLOC: Linux 58
15 Commits per author, posts per poster and bugs per submitter: Blender 59
16 SLOC evolution and subsystem growth: Blender 60
17 Commit and post frequency: Blender 61
18 Bug arrival rate and arrival rate per commit rate: Blender 61
19 SLOC per NOM and CYC per NOM: Blender 62
20 Fan-in and fan-out per SLOC: Blender 62
21 Information flow, DIT and NOC, and CBO, WMC, and fan-in: Blender . 63
22 Commits per author, posts per poster and bugs per submitter: GIMP 64
23 SLOC evolution and subsystem growth: GIMP 65
24 Commit and post frequency: GIMP 66
25 Bug frequency and arrival rate per commit rate: GIMP 66
26 SLOC per NOM and CYC per NOM: GIMP 67
27 Fan-in and fan-out per SLOC: GIMP 67

List of Tables

1 Knowledge requirements in software quality measurement 4

2 Halstead's software science indicators 13

3 Object-oriented metrics by Chidamber and Kemerer 16

4 Object-oriented metric hypotheses by Ferenc et al. 34

5 FOSS information spaces and data sources 40

6 Data sources for experiment 47

7 Number of messages and time to import them 47

8 Number of bugs and time to import them 48

9 Number of revisions, run interval for metrics calculation, and time to import them 49

10 Description of calculated metrics 52

11 Developer classes: Linux and Linux-historical 54

12 Developer classes: Blender 59

13 Developer classes: GIMP 64

14 Summary and interpretation of experiment results 68

1 Introduction

Free and Open Source Software (FOSS) has attracted increased attention in the computer software industry. Producing computer software is a complicated and laborious task. To people familiar with some of the well-known, rigid process models used in the industry, it may seem surprising that such a loosely defined work method can produce any results at all. Meanwhile, FOSS projects have produced a staggering amount of software, ranging from simple, single-purpose tools to complete operating systems, server applications, and desktop suites. FOSS is now a serious competitor in several market segments, and is likely to change the business patterns of the software industry [Lin04].

FOSS emerged from a culture of hobbyist and professional computer programmers [HaO01, Lin04]. It was not originally a defined concept; it was simply the way in which these pioneers found themselves practising their activity in an environment with active exchange of ideas and program code, and strong ties to an academic research environment. As the commercial software market developed, the concepts of Free Software [Fre07] and later Open Source [Ope07] were established to capture two important parts of that culture. Understanding them is vital for anyone who wishes to observe or participate in FOSS [Lin04].

In FOSS development, quality and measurement of quality-related attributes are just as important as in conventional software development, and is unfortunately neglected at least as often. However, the value system of FOSS development often emphasizes different aspects of quality due to different underlying assumptions and a different working method. The differences may be subtle in some cases and more explicit in others.

Since FOSS projects are typically carried out using publicly accessible mailing lists, documentation, and source code repositories or version control systems, the software is in a constant state of change and every change is visible to all. It may be difficult to grasp the current state of the software and to evaluate its suitability for a particular use. On the other hand, since the information is available without restriction, it is possible to make quite an elaborate and accurate evaluation. Care must be taken, however, not to misinterpret the findings or compare them without taking the intended state of development into consideration.

Emphasis on quality is important in a FOSS context for several reasons. Customer expectations determine success in the marketplace, and competition is less and less restricted by copyright but instead based on competence [Lin04]. At the same time, competence among FOSS project participants varies greatly and as a result, quality is not reliably managed. The ability to understand quality, to measure it, or to take appropriate steps to assure it are not among the frequently found skills in FOSS projects. However, interest in the subject is slowly rising.

This thesis explores the relationship and tensions between existing systematic quality practices and the diverse and often ad hoc practices of the FOSS community. It is our belief that the FOSS community has a will to improve software and process quality beyond what is currently achieved, and that it will assimilate knowledge and methods that are practically usable. Also, entities wishing or needing to observe or participate in FOSS development or use can benefit from knowledge of how to apply and interpret quality assessment in a FOSS environment.

The rest of this thesis is organised as follows. Section 2 describes how quality, a subjective concept, can be quantified and measured by identifying a value system, choosing a quality system, decomposing it into discrete metrics and calculating those in a given piece of software. It also introduces some existing metrics and models of quality. Section 3 briefly explores the different value systems that are applicable to FOSS, and then proceeds to discuss quality in a FOSS context. Finally, it presents a number of metrics and metric models. The section approaches these themes from the perspective of existing work on the subject. Section 4 presents and classifies a set of metrics that are useful for FOSS projects. At the end of the section, a taxonomy is constructed that describes the overall quality of a FOSS project and provides drill-down capability into the components of this view of quality. The section explains why it is important to choose metrics that are easy to understand and can, as far as possible, be collected automatically. Section 5 describes the experiment environment and applies the taxonomy constructed in Section 4 to a number of real-life FOSS projects. Section 6 presents the results of the experiment described in Section 5. Section 7 presents conclusions and findings based on the entire thesis, and summarizes central concepts.

2 Quality as a measurable quantity

According to the IEEE Standard for Software Quality Metrics Methodology (SSQMM), quality is “the degree to which software possesses a desired combination of attributes” [IEE04]. Boehm et al. define overall quality as some function of metrics that provide "a quantitative measure of the degree to which the program has the associated characteristic" [Boe76]. These are very high-level definitions that do not yet specify how to actually measure quality in any amount of detail. Their application is much more complicated than what might be inferred from these simple, one-sentence definitions.

2.1 The intuitive view of quality

Intuitively, software quality is about a computer program working "as it is supposed to". Customer satisfaction is a key ingredient [Roy90, MäM06]. In other words, there is an element of expectancy that the program does not act in ways which surprises the user. For example, a particular user may feel that the program must not carry out its actions too slowly, it should not produce incorrect results, and it should not crash or lock up. Another user might not care as much about the speed of the program, as long as its result is always very precise. However, user expectations may be misleading; yet another user may be accustomed to bad performance and reliability, and is not surprised at all if the program works only randomly. In this case, quality will in fact deviate from user expectations.

Also, the intuitive view of quality might include requirements that the program is easy enough to use, and could also include matters of taste and habit, such as the colour scheme used or the particular order in which certain actions are performed; local law and culture, such as the ability to retain call data in a telecommunications system or to protect the privacy of a health care client; or the particular needs of a single organisation or group of people.

The intuitive view of quality is useful, because it is a starting point from which to approach a definition of quality. In the IEEE SSQMM, the establishment of software quality requirements starts with an intuitive view of quality, from which the requirements and constraints are drawn:

> Use organizational experience, required standards, regulations, or laws to create this list. ... Consider contractual requirements and acquisition concerns, such as cost or schedule constraints, warranties, customer metrics requirements, and organisational self-interest [IEE04].

It should also be noted that quality at this level is fundamentally a patchwork of conflicting needs. As Boehm et al. state:

> ...added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness can conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations [Boe76].

Similarly, Schneidewind lists several areas of knowledge that are required in software quality measurement [Sch02]. Quality measurement can benefit from multidisciplinary input, including different types of engineering, economy, and mathematics (Table 1). Pinning down the intuitive view of quality is not straightforward and it must be done in several overlapping ways simultaneously. Software engineers, Schneidewind argues, must be fluent in several of these areas in order to understand "*why* and *when* to measure quality". Why to measure quality is directly associated with the chosen value system, whereas when to measure quality is a matter of knowing how to tie quality measurement into the overall software development process.

An intuitive view of quality concerns not only the user, but also the designer and writer of software. To the designer, conceptual elegance is sometimes considered high

Issue	Function	Knowledge
Goals	**Analyse** quality goals and **specify** quality requirements.	Quality engineering, requirements engineering
Cost and risk	**Evaluate** economics and risk of quality goals.	Economic analysis, risk analysis
Context	**Analyse** the applications environment.	Systems analysis, software design
Operational profile	**Analyse** the software environment.	Probability and statistical analysis
Models	**Model** quality and **validate** the model.	Probability and statistical models
Data requirements	**Define** data type, phase, time, and frequency of collection.	Data analysis
Types and granularity of measurements	**Define** the statistical properties of the data.	Measurement theory
Product and process test and evaluation	**Analyse** the relationship between product quality and process stability.	Inspection and test methods
Product and process quality prediction	**Assess** and **predict** software quality.	Measurement tools

Table 1: Knowledge requirements in software quality measurement [Sch02].

quality. A well designed class hierarchy does not break or lose its logical composition when introducing new classes as the program evolves. The task of programming according to specifications becomes possible, and there is little need to redesign the program. Programming tasks are easily divided among the programmers. For example, the program can be written one class at a time and one method at a time, relying on the designed interfaces. Even if one module is changed, the change stays limited and does not cause a need for further change in other modules. There is no risk of incompatibility, because the design is correct and of high quality. It fits into the overall production process and enables product quality.

To the writer of software, quality goes beyond the design and into the lowest levels of the program. Each line of code can have an aesthetic quality. It can be easy to understand, making the code fluent to read and thus easy to modify. However, other constraints, such as efficiency requirements, might lead to code that is hard to read but performs exceptionally well. This is also an aspect of quality, as is the ability to write terse, compact code – including as much functionality as possible in as few statements as possible. It is impossible to dismiss any such view of quality without first establishing what is to be accomplished.

To both designers and writers of software, it seems that ease of change, or *flexibility*, is a key quality criterion on the intuitive level [Roy90]. It is a prerequisite for ongoing development. Unless the software can be practically changed, there is a real risk of financial loss to users and developers. We observe that flexibility is one of the key motivators behind the structure of many FOSS projects.

This intuitive view provides little or no possibility to assess whether or not a software product is of high quality, but it provides the value system from which a more formal definition of quality can be drawn. In other words, there exists no universal definition of quality. Rather, each application of quality methods must take into account the particular setting in which the software is to be used and maintained. By analysing quality goals, requirements, and context; and assessing the cost and risks of these, a software project can determine what the value system of the users is. Fortunately, many value systems have a lot in common, and therefore quality practitioners can reuse existing work, enabling comparison of software and development processes.

Since quality is such a multi-faceted concept, a rigorous definition is needed to remove the ambiguities and allow a group of people to work toward the same, known quality goals [Sto90]. This view of quality must be measurable, otherwise it is impossible to know whether or not quality has been achieved.

2.2 Translating intuition into a quality system

Given an intuitive view of quality, it seems obvious that choices must be made to remove ambiguity and balance conflicting needs if one is to systematically approach quality. Several methods have been proposed, some of which are completely theoretical and others which are based on established industry practice.

Boehm et al. reported their findings in 1976. Their work has been ground-breaking in the software industry. They describe results from an earlier study that was to identify a set of characteristics of software quality. Initially, they set out to define a single overall quality value based on a combination of metrics:

> 1. Given an arbitrary program, the metric provides a quantitative measure of the degree to which the program has the associated characteristic, and
> 2. Overall software quality can be defined as some function of the values of the metrics [Boe76].

However, after closer evaluation, they concluded that "there is ... no single metric which can give a universally useful rating of software quality". They proceeded to describe one possibility for breaking down quality into a hierarchical set of characteristics and a set of anomaly-detecting metrics. The goal was no longer to provide a single model in which all software could be assessed, but rather to provide a conceptual framework in which each software project could find guidance in establishing their own software quality practice.

They used a stepwise approach to define quality characteristics and metrics attached to those. In each step, they manually refined the characteristics and metrics to avoid overlap and increase coverage. They found that the characteristics were related in a type of tree structure (Figure 1). The lower levels of the tree are preconditions for their parent characteristics, so for example, the degree of *understandability* and *testability* directly influence the degree of *maintainability* of the program. Thus, the tree reflects the value system that they chose. Also, the idea of a single, overall quality value was not abandoned. The desire was to have a quality system that would allow some degree of comparison between different software products.

It is interesting to note that the view of software held by Boehm et al. is similar to the FOSS view in one important way: it implicitly assumes that whoever acquires the software has access to the source code. In fact, software seems to be equated with source code. At the top of Boehm's tree of quality is the characteristic *general* *utility*. This includes, but is not limited to, the answer to three questions which Boehm claims are the main questions when acquiring a software package [Boe76]:

- How well (easily, reliably, efficiently) can I use it as-is?
- How easy is it to maintain (understand, modify, and retest)?
- Can I still use it if I change my environment?

Software is often distributed in compiled object form only, and the source code is not provided. Thus, the notion of general utility must have changed, or software quality has been significantly obscured, because there is generally no way to understand, modify, or retest the binary form of a program with the purpose of maintaining it, and it can generally not be used if the environment changes – recompilation would be necessary. Of course, this only serves as evidence that the underlying value system is decisive when assessing quality. The software industry was different in 1976. Software was not yet as pervasive a consumer product as it is today. Boehm et al. operated in a specific environment, but some aspects of that time and environment have been retained in FOSS.

Boehm's tree of quality is problematic in other ways as well, as is any model of quality that is more or less strictly arranged in a similar fashion. While it does provide

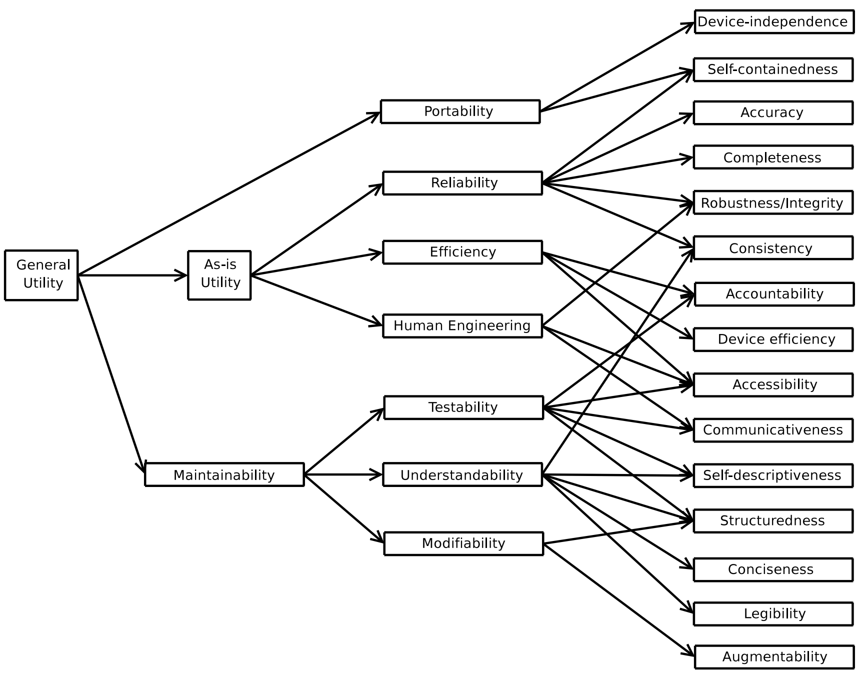


Figure 1: Software Quality Characteristics Tree [Boe76].

insight into the quality characteristics and their relations, it does not accurately define them. The final and decisive definition becomes an exercise in interpreting ambiguous human language, instead of being defined in an objectively measurable way. Also, the notion of an overall quality measure is extremely difficult to apply in practice, and could at best be used in a limited subset of software with very similar requirements [Sto90].

Even so, the notion continues to be part of many definitions of quality systems. The IEEE SSQMM describes a flexible framework that helps an organisation to divide the view of quality into meaningful parts, quality attributes, which describe the quality of the software system that is being built (Figure 2). The quality attributes are then assigned quality factors, which can in turn be assigned subfactors. Finally, factors and subfactors are associated to metrics that serve as quantitative, measurable representations of the factors. It should be noted that there is no assumption or claim in this framework that the metrics or factors can be recombined to give one single quality value. Rather, the framework serves as a conceptual aid to establish which facets of quality the organisation wishes to monitor.

The task of upholding a quality system in an organisation is called *quality assurance*. It is performed by a department or organisation that defines the standards for quality, specifies and sometimes implements tools and aids for assessing quality, and applies the tools to the software created by other parts of the organisation in order to check for adherence to quality standards, to give them feedback, and to suggest areas of improvement [Gaf81]. This function is commonly folded into the program-

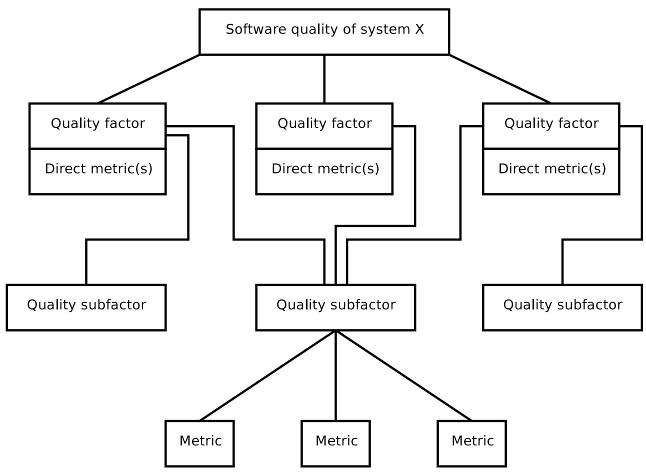


Figure 2: IEEE software quality metrics framework [IEE04].

ming team in order to make communication more effective or when resources are scarce, but it may also be an external function.

In order to translate intuition into a quality system, intimate knowledge of several different issues is required, and they must be balanced to fulfil the requirements in each particular situation. As the requirements change over the lifetime of the software, it may be necessary to revisit the choices and adjust the balance. Therefore, it is important to retain some knowledge of the underlying value system and the choices made in the translation to a quality system.

2.3 Metrics as the building blocks of quality measurement

Software metrics are measurements of some properties of a software system, its specifications, and the software project to develop it [Cha06]. Some possible uses of software metrics are to estimate the time required to build a certain piece of software, the cost of the same, the number of programmers required given certain time constraints, or, in the case of software quality metrics, to establish whether the software meets the defined quality requirements and to alert an ongoing project of emerging quality issues.

As indicated in the previous section, there is both theoretical and practical support for a general view of quality measurement that relies on metrics. The complete picture of quality is broken down into pieces, possibly in a hierarchical fashion and in multiple steps. Each piece is a measurable attribute of the software. The measurements can be divided into *static* or *internal* and *dynamic* or *external*. The former is derived from the program itself, such as the source code. The latter is derived from the behaviour of the program, such as assessment of the running program or an aspect of running it [Gaf81, Nag04, Nag05]. In general, we wish to influence the external aspects, but we are limited to affecting the internal ones. Internal metrics can be collected earlier and easier than external ones, and have been shown to correlate with externally visible quality [Nag04, Nag05].

The purpose of metrics is both descriptive and predictive. In their descriptive capacity, they help to classify software and projects. For example, the size of the source code, the number of developers, the programming language used, and the age of the software product are useful classifiers.

In their predictive capacity, metrics assist in performing educated guesses and support decisions that affect the future of the software or project. For example, field defects are defined as user-reported, code-related problems requiring programmer intervention to correct. Predictive metrics can be used to estimate the number or likely location of the field defects, so that their number can be minimised before the software is deployed and put to use. This implies that some metric data must be available before the software is finished.

When assessing quality, samples are taken from the software and used in a software *metric model*. In general, the model relates the measured attributes – *predictor* metrics – to a *target* metric. The target metrics are quality factors as specified by the quality system. Some models attempt to give an overall quality value by taking all factors into account and relating them to a single target factor, while others simply highlight different aspects of quality.

Once a metric model has been constructed, it is important to validate it, both theoretically and with empirical data. The purpose of validation is to ensure that our measurements are in fact giving us the information we seek. One part of validation is to run the metric model against a data set that has been validated by human observation, and see how well it predicts the known outcome in those cases. For stochastic models, the prediction accuracy may be expressed as an error margin. Another part of validation is to theoretically confirm that a metric is an accurate representation of the phenomenon it measures. Knowledge of statistical and measurement-theoretic concepts is needed in validation (Table 1).

Comparing metric models of different kinds can lead to adoption of valid models with better accuracy. Khoshgoftaar et al. have compared regression modelling with neural network modelling in software quality models. Initially, they observe that the set of available metrics is large, and that the metrics are often highly correlated. This is not desirable in multiple regression modelling. On the other hand, it is beneficial for neural network models. In those, multiple inputs should lead to a more complete model as the metrics focus on different, but overlapping and complementary aspects of the software. In their comparison, Khoshgoftaar et al. find that neural network models yield more accurate results although they require more time to train than regression-based methods [Kho95].

Not all metrics are important for every case, nor is there likely to exist a single technique that suits all cases [Ola92]. To establish a predictor as important, Li et al. list three methods [Li05]. First, high correlation between the predictor and field defects can be shown. Second, it can be shown that the predictor is selected using a model selection method. Third, it can be shown that the accuracy of predictions improves with the predictor included in the prediction model.

A great number of predictor or metric models and related metrics have been constructed, described, and validated over more than 30 years. At the same time, software metrics are not put to good use in mainstream software engineering, partly because of their complexity but also because their relation to practical decision-making is vague [FeN00]. Metrics are often used as simple passing criteria for a specific process phase, typically the testing phase. From a quality perspective, it is not adequate to simply count the amount of automated tests that the software passes, or the number of defects discovered by the tests – quality is built into every action during the entire software production process [Oga96]. This is also important because quality assessment must be performed early enough to allow developers to take corrective action [Nag04, Nag05].

2.3.1 Analysis and design metrics

Analysis and design metrics, also known as structure metrics, are global indicators of quality which can be taken early in the software life-cycle. They measure those features of the software product that have come into existence during the analysis and design phases [Kaf85]. Thus, they function as early indicators of the potential quality of the final product, and also as a basis for effort, cost and risk estimation.

The primary use of analysis and design metrics is before the implementation phase starts. If the design quality is too low, the project may have to take a few steps back and revisit some design decisions [Roy87]. Missing knowledge might be a reason for bad design, and that could be alleviated by choosing an iterative process model that does not assume that all the required knowledge is in place from the start. Structure metrics can also be used to compare the potential quality inherent in the software design with actual quality in the final product.

The *information flow metric* is a structure metric that measures the sources (*fan-in*) and destinations (*fan-out*) of all data related to a given software component. These factors are used to compute the communication complexity of the component, which is taken as a measure of the strength of its communication relationship with other components. The fan-in consists of all function parameters and global data structures from which the function retrieves its information, while the fan-out consists of the return values from function calls and the global data structures that the function updates [Kaf85, WaH88].

Another important structure metric draws attention to the "ripple effect" that occurs when a change to one component causes a need for change in other components. In this *stability* measure, the flow of data through parameters and global variables is used to identify the components which could be affected by a change in a particular component [Kaf85].

Both of these metrics reflect a quality risk. If program components are designed to be too dependent on each other, flexibility is lost. This means the program is more difficult to change in the future.

Other metrics that can be taken once the analysis and design phases are complete are the number of documents produced, the size of these documents, and several metrics that analyse the contents of the documents. Using these, a project could determine whether the analysis and design quality is sufficient, or if more work is needed.

2.3.2 Code metrics

Code metrics are all those metrics that measure the actual code produced in the implementation phase of a software process. Together with structure metrics, they belong to the class of internal attributes of a program. Code metrics are problematic in the sense that they are available only after the code has been produced, and thus it is relatively expensive to correct the problems they might reveal compared to addressing the issues in the design phase [Kaf85, Roy87]. When simply measuring software quality, they are of course useful regardless of whether addressing them is practical or not.

Perhaps the oldest and most obvious metric is the *lines of code* metric (LOC). It is a measure of program size, and although simple at first glance, there are some issues to consider when defining the precise way of calculating it, such as whether blank lines and comments should be included. LOC is a basic metric that is frequently a component in several higher-level calculations. For example, programmer productivity measures and quality criteria have used LOC as their basis. LOC has received criticism, but it has been found no less valid than other, more sophisticated size measures [Kaf85].

Halstead's *software science* is a collection of metrics and equations based on the number of unique operators and operands, and the number of total operators and operands in a program [Hal75]. From these, Halstead derives measures of program size, level of effort, programming time, and other measures of the software and implementation language (Table 2). Some studies support Halstead's equations and empirically find correlation between, for example, effort level and number of defects, indicating that the metrics may have an application in quality measurement. However, other studies find no support for the equations [Kaf85, Coo82].

Basic parameters	Calculated metrics	
n_1 unique operators used	**Program length**	$N = N_1 + N_2$
n_2 unique operands used	**Vocabulary**	$n = n_1 + n_2$
N_1 total operators used	**Volume**	$V = N \times log_2(n)$
N_2 total operands used	**Effort indicator**	$E = \frac{V}{\frac{2}{n_2} \times \frac{n_1}{N_2}}$

Table 2: Some of Halstead's software science indicators. Adapted from Halstead [Hal75] and from Wake and Henry [WaH88].

Cyclomatic complexity, defined by McCabe, is a count of the number of decision points in a program. The number is based on a graph representation of the program. The formal definition is $v(G) = e - n + 2$, where e is the number of edges in the graph, and n is the number of nodes. McCabe has shown that this number relates to the cyclomatic number of the graph representing the control flow of the program, and he further associates this with ease of testing the program [Kaf85]. However, it is not entirely clear that the cyclomatic complexity corresponds to perceived, or psychological complexity, even though it is often interpreted as a measure of comprehensibility [Kaf85, Coo82]. As with the Halstead metrics, an increase in this metric could indicate quality issues. The program might be complex and hard to maintain. However, the implemented algorithm also has an inherent lower complexity bound, so it is not possible to decrease complexity beyond a certain lower limit.

2.3.3 Object-oriented metrics

Since the rise of object-oriented programming languages, a number of researchers have attempted to apply traditional software measurement on object-oriented programs. However, this has been criticised both because of the lack of a theoretical foundation of the traditional metrics and because object-oriented programming may be fundamentally different when it comes to problem-solving behaviour. Therefore, a set of metrics designed specifically for object-oriented software is justified. Because the abstraction level of object-oriented languages is different than that of procedural languages, design of object-oriented systems produces designs that are closer to the implementation code. Therefore, these metrics can be partly used to measure both design and implementation – although some of them can only be used when the implementation exists.

Chidamber and Kemerer have developed object-oriented metrics that have a rigorous theoretical foundation and have validated them empirically [ChK94]. Their approach focuses on the design of object-oriented software, and they argue that the benefits of design evaluation can be substantially greater than metrics aimed at later phases of the software life-cycle. Their metrics are independent of the particular object-oriented language. This means that upon actual use, some choices must be made depending on the language in question.

The authors draw from theoretical philosophy in defining the following terms:

Coupling: The degree of interdependence between parts. Two objects are coupled if at least one of them acts upon the other.

Cohesion: The internal consistency within parts. In object-oriented programming, the degree to which related things are kept together; similarity between methods. Similarity between methods can be seen as the intersection of the sets of instance variables used by the methods.

Complexity: The larger the number of properties, the higher the complexity. It can be defined as the cardinality of the set of properties of an object.

Scope: How far the influence of a property extends in the class hierarchy. The influence of a property on descendant classes is indicated by the number of children of the class that has the observed property. The influence from properties of ancestors of a class is indicated by how deeply into the inheritance tree the observed class is located.

Combination: The result of combining two or more classes to generate another class. This is related in some programming languages to multiple inheritance, but also to class hierarchy, and the result is that the obtained class has as its properties the union of the properties of the component classes.

Using these definitions, which the authors define precisely using set-theoretical concepts, a number of object-oriented metrics are given (Table 3). *Weighted methods per class* (WMC) is the sum of the complexities of all methods in a class. Chidamber and Kemerer purposely do not define how the complexity of a method is calculated, to avoid making this metric specific to any programming language. They suggest that some traditional metric may be appropriate. The utility of this metric is that it predicts the time and effort required to develop and maintain the class. With increased WMC, the potential impact on the children of the class is increased, which means the class may have a greater impact on the quality of the program. Also, the possibility of reusing the class is decreased due to specialisation.

Depth of inheritance tree (DIT) is the maximum length of a path from the node representing a class to the root. This metric relates to the scope of the class, and it measures how many ancestor classes may potentially affect the class under observation. With increased DIT, the complexity of the class is likely to increase as there are more possible ancestors to inherit methods from. Also, a large overall DIT means more classes and methods and thus greater design complexity. The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods. All these factors have potential quality implications.

Number of children (NOC) is the number of immediate descendants of the class under observation. This also relates to the scope of the class, and measures how many subclasses inherit the methods of the class. With increased NOC, the reuse through inheritance has increased, but the likelihood of subclassing misuse has also increased. NOC is also a measure of the potential influence the class has on the overall design.

Coupling between object classes (CBO) is the count of the number of other classes to which the class under observation is coupled. Coupling between two classes exists when one of them uses methods or instance variables of the other. A lower CBO means the class is more independent and has greater potential for reuse. Also, a high overall CBO means greater sensitivity to changes, increased maintenance difficulty and greater need for testing.

Response for a class (RFC) is the number of methods that can potentially be invoked in response to a method being called from outside the object. Increased RFC means the class is more complex and testing and debugging is more complicated.

Lack of cohesion in methods (LCOM) relates to the similarity between all method pairs in the class. The similarity is determined by the number of instance variables that both methods use. Decreased LCOM means the class is more cohesive, which promotes the desired trait of encapsulation. Increased LCOM increases complexity and implies that the class should probably be split into two or more subclasses.

Chidamber and Kemerer argue that a well architected system will have a stable class hierarchy, which will be reflected in the WMC, NOC and DIT metrics over time. RFC and CBO metrics can be checked to detect if there are unwarranted connections between classes. If the definitions of the classes themselves are changed, WMC and LCOM metrics will reflect this. Thus, a manager can use these metrics to gain an evolutionary overview of the implementation and maintenance quality of an object-oriented program.

Abbreviation	**Name**	**Description**
WMC	Weighted methods per class	$WMC = \sum_{i=1}^{n} c_i$, where $c_1, \ldots, c_n$ are the complexities of each method in a class.
DIT	Depth of inheritance tree	The maximum length of a path from the node representing a class to the root.
NOC	Number of children	The number of immediate descendants of a class.
CBO	Coupling between objects	The count of the number of other classes to which a class is coupled. Coupling exists when one class uses methods or instance variables of the other.
RFC	Response for class	$RFC = \|RS\|$, where $RS = \{M\} \cup_{\forall i} \{R_i\}$. $\{M\}$ is the set of all methods in the class and $\{R_i\}$ is the set of methods in any class called by method i in the observed class.
LCOM	Lack of cohesion in methods	$LCOM = \begin{cases} \|P\| - \|Q\| & \text{if } \|P\| > \|Q\| \\ 0 & \text{otherwise} \end{cases}$, where P is the set of method pairs whose number of shared instance variables is zero, and Q is the set of method pairs whose number of shared instance variables is not zero.

Table 3: Object-oriented metrics as proposed by Chidamber and Kemerer [ChK94].

2.3.4 Maintenance metrics

Software maintenance is an extremely important part of the software life-cycle. Estimates of the relative amount of resources spent on maintenance during the entire life-cycle range from 40% to 67%, and the cost of fixing a defect increases as the software life-cycle progresses [WaH88, Roy87]. The value of high quality software is thus at its peak during the maintenance phase, and it is therefore economical to try to predict the maintainability, and correct defects that affect maintainability as early as possible in the software life-cycle. Since the maintenance phase can continue for an indefinite amount of time, metrics that are usable during that phase are valuable.

Visaggio's *structural information* ($I(v)$) metric [Vis97], is an example of further development of the "ripple effect" idea, introduced in Section 2.3.1. It attempts to connect decisions made in the requirements analysis, design and implementation phases of a software project. Visaggio constructs a dependency graph that shows how decisions in the analysis phase have resulted in design artefacts in the design phase, and how these in turn have resulted in specific parts of the implementation. The metric then measures how a change in one part of the system will affect other parts. Visaggio relates this metric to the notion of quality of organisation in the software.

Visaggio lists several advantages of this metric compared to what he calls "the common complexity metrics" which are presumably code metrics such as those described by Halstead and McCabe (Section 2.3.2). The metric has not been empirically validated, and its biggest practical weakness is that the dependency graph cannot be built automatically, because the necessary tools to produce machine-readable artefacts in all stages of development are not commonly available. Such tools would have to be used from the earliest point in the process, and could make the design work more tedious and less flexible.

As another example of a metric model that predicts quality in the maintenance stage, Sharma and Jalote define *stabilisation time*, a metric that measures how long it takes for a software product to reach a steady state of defect arrival frequency after it has been installed [ShJ06]. They note that the failure rate often declines after software installation, and then stabilises at a certain level, after a certain time. Possible causes for this are that users learn to work around the defects, that configuration issues in new installations are resolved, and that user experimentation ceases after some time. This metric could be applied to FOSS, with proper adjustments to fit each project.

In general, metrics for the maintenance phase are the same as those for other phases, but instead of taking single samples at the end of each development phase, regular samples are taken and the rate of change is noted. Maintenance metrics can be thought of as software evolution metrics.

2.3.5 General metrics problems

One frequent problem with software metrics, which is accentuated in quality metrics, is that of incomplete data or missing samples. Chan et al. have applied statistical methods to such data in an effort to reduce the impact of this missing data. They used an imputation method to fill in missing data, which proved successful in their case study. Also, they have tested a formal method to verify the statistical significance of predictor metrics on the target metrics and to eliminate unnecessary categories in categorical predictor metrics [Cha06].

Another problem, prominently present in FOSS development, is the lack of software engineering experience, easily understandable metrics, and general understanding of quality models. This leads to an inability to comprehend and make use of quality improvement methods, or to participate in projects where such methods are used. Conversely, FOSS projects already using such methods have a higher barrier of entry and may fail to attract the needed number of programmers.

Houdek and Kempter propose a method to address this issue in any software organisation [HoK97]. Their aim is to explore how experience can be collected, stored and disseminated without distortion – written material is often ambiguous and incorrectly interpreted. They wish to use the findings for quality improvement. They observe that the ability to understand why a software project has succeeded is key to a repeatable process. They propose to use a systematic, scientific method to gain insight into the process and then package this insight for reuse in later projects.

The scientific method of systematic information collection is an important step, and its primary objective is to describe the methods involved. However, Houdek and Kempter emphasise that the insights gained through observation must be disseminated or packaged in a reusable form. With this in mind, the authors propose the use of *patterns*, pairs of problem and solution that can be applied, with some modification, to a new problem. Similarly to design patterns, which guide the design and implementation of software, quality patterns would be employed to guide quality assurance.

Another problem is that statistics and machine learning can be very challenging to interpret. The methods produce results, but the underlying model is often incomprehensible to humans. Developing ways to visualise metrics can help observers who do not have the time or knowledge to interpret complex, number-heavy statistical models to extract the underlying rules. Also, if data exploration is desired, the choice of one particular model may be premature and visual presentation beneficial.

Langelier et al. have experimented with mapping metrics to three-dimensional visual objects. They mapped the size, cohesion and coherence of classes in object-oriented programs to the size, colour and rotation of three-dimensional cubes and arranged them together to form a map of the program [Lan05]. This allows the observer to gain an overview of the software, and assists in choosing more specific features for closer observation.

Liu et al. emphasise that practical exercise in using quality prediction models is important in education [Liu07]. They describe an experiment set-up with prepared data that can be used to train students in the use of quality metrics. There is an abundance of FOSS documentation that describes the use of software and participation in projects in a *how-to* or pedagogical fashion, but we have not found any such document regarding FOSS quality. This is an area where FOSS could improve significantly.

3 Quality in the context of Free and Open Source Software

In the philosophical and reflective works that define FOSS, the motivation of practitioners is idealistic, regardless of whether their primary focus is ethics or economics. This value system concerns the potential use and re-use of the software, not its functional aspects. At the same time, other FOSS advocates often cite higher quality in technical form as the primary practical benefit. Defects, or bugs, will be eliminated because of the sheer number of people looking at them – this is the reasoning often presented. Peer review could be considered the primary method of quality assurance in FOSS projects. It somewhat resembles both the academic notion of peer review [Mic05a] and a formal technical review of program code, but may take many forms in practice [MäM06].

However, simply labelling a piece of software "FOSS" does not increase its technical quality. In fact, most FOSS projects struggle with the same kinds of quality and process issues as most traditional software projects. Controlling functionality, efficiency, reliability, usability, maintainability, and portability of software is difficult. The challenge to deliver a product on schedule and according to specifications is not automatically met by placing the source code in a publicly accessible repository.

In fact, FOSS may be completely overlooking certain aspects of software quality simply because the notion of quality has not been defined for FOSS. Developers may be eager to solve their own problems and cater to their own needs but believe that all others are able or inclined to do the same. Quality is often equated with the number of discovered defects that have been removed, and it is believed that a vast number of random tests will efficiently remove all defects. Similarly, it is often said that FOSS will automatically result in more secure software [Gre03] – again assuming that source code availability will ensure this. Not all advocates emphasise these aspects of quality; some focus on non-technical benefits instead. But if FOSS is to be a repeatable practice, more insight into the reasons for success and failure is needed.

3.1 The FOSS mind set: catalyst or obstacle?

We observe that FOSS projects can quickly adapt and change their working methods, but that they are surprisingly resistant to certain types of change. Also, FOSS developers usually have a very specific picture of themselves and their projects. Does this support or undermine quality assurance, and how do FOSS projects lend themselves to quality assessment? We will explore these questions by presenting a detailed account of what FOSS development is or can be, and what it is not.

In an analysis of exploratory interviews with seven FOSS developers, Michlmayr et al. present the mind set of the developers and describe some key characteristics of FOSS projects, such as formation, membership, and work processes [Mic05a]. They tie the analysis into a question of quality: what is the current understanding of quality in FOSS projects, and how could these projects improve quality in both their work processes and in the final software deliverable?

Michlmayr et al. draw attention to two FOSS project characteristics that have important implications on quality assurance: their distributed nature, and the fact that the participants are usually unpaid volunteers. Project-level practices that are commonly referred to as *human resource management* are difficult to achieve, since the participants cannot easily be held accountable for their involvement in the project. For instance, it is difficult to delegate tasks that no-one volunteers to perform. The project can only do things that a contributor is prepared to carry out. In a FOSS project with paid employees, the mechanism for affecting motivation is different, but the topic of how to motivate participants is different from the question of whether the work is performed or not.

A common pattern that Michlmayr et al. present as background to their research is that a multitude of FOSS projects are abandoned early, and only relatively few have a large number of participants. Michlmayr et al. note that “more interesting projects with a higher potential will probably attract a larger number of volunteers” but also that project failures might be related to lacking project management skills [Mic05a]. Thus, it appears that a project needs both skilled management and a “critical mass” of contributors to be sustainable.

Although only including comments from seven developers, the interviews appear to have captured some of the beliefs that are commonly found among FOSS developers. The interviewees were asked questions about quality management in FOSS and non-FOSS projects, and otherwise the interviews were allowed to take a free form. Thus, the interviews attempted to capture a subjective or intuitive perception.

The interviewees presented the opinion that FOSS has a "higher potential to achieve greater quality" and "can react faster to critical issues" such as security flaws, compared to non-FOSS software. This is attributed to a number of factors such as *more feedback* in the form of bug reports and feature requests, *higher motivation* because contributors participate by their own free choice, and chance of *attracting better human resources* because the projects are distributed in nature and can draw from a greater pool of knowledge and expertise. The authors note that it is difficult to compare FOSS software with non-FOSS software, because there is seldom access to source code and defect reports from the latter. The authors do not report on the interviewees' overall software engineering experience.

The authors note very varied practices among different FOSS projects. For instance, the infrastructure varies: projects use many different tools for defect tracking, communication, and source code storage and versioning. Some projects have rigorous requirements that must be fulfilled before committing new or changed source code into the source code repository, while others have more relaxed requirements. Some projects allow nearly anyone to commit directly into the repository, some have entry requirements and others allow only a small, fixed number of developers to commit while other contributors have to submit patches for review and possible later inclusion. Naturally, this is also affected by the capabilities of the source code repository system.

Michlmayr et al. note a number of process-level practices that differ between projects. Projects can differ in the way they allow new participants to *join*, what requirements they impose upon *releasing* a new version of their software, how they create and manage different versions or *branches* during development and after release, how *peer review* is performed, if at all, how *testing* is organized, and finally, what overall provisions the project has in place for *quality assurance*.

Furthermore, different kinds of projects have different kinds of goals and may require different kinds of leadership [Nak02]. Nakakoji et al. have explored evolution patterns in FOSS systems and communities, and draw attention to three types of FOSS projects. Exploration-oriented projects wish to share innovation and knowledge and have an individual, centralised style of control. Utility-oriented projects aim to satisfy individual needs and have a decentralised style of control. Service-oriented projects attempt to provide a stable service and have a central governing body with a small number of members.

3.1.1 How FOSS compares to process models

It is easy to think that FOSS is a process model. It would fit conveniently beside traditional process models, such as the waterfall model, and iterative or change-embracing models, such as Agile software development models. While there may be similarities, and while FOSS may apply some of the same steps and procedures, there is too much variance in FOSS projects to support the claim that all FOSS projects would share a common process model or meta-model. FOSS projects may employ any process model, or none at all. However, FOSS does have a direct impact on the process employed by a project, whether or not that process is consciously or rigorously defined.

Some of the main characteristics of FOSS are posed as differences compared to traditional, industrial software development. Mockus et al. list a number of such differences [Moc02]. FOSS is built by potentially large numbers of participants, ranging in the hundreds or thousands. Work is not assigned, participants choose their own work. There is no explicit system-level design, or even detailed design, no project plan, schedule, or list of deliverables.

However, Mockus et al. may not have looked carefully enough. They observe that the chance of success increases if developers use the software they write. If so, the requirements and design exists privately with the developer, although it is not communicated. In other cases, and we believe this is increasingly common, FOSS projects do have plans, schedules and other project-level documentation [Mic05c], but they are part of a continuous design effort that runs in parallel with the rest of development.

Mockus et al. also explicitly distinguish “pure” FOSS projects, which they define as those that have no significant commercial involvement. This is a highly problematic label, and it is not clear what its utility is. First, commercial involvement does not appear to select the development methodology or process, and second, it may be impossible to determine which FOSS projects do have a commercial involvement and which do not – individual developers may benefit financially although the project does not charge fees for the software. Third, the notion of *significant* commercial involvement would have to be based on the real impact of the involvement, which Mockus et al. do not define.

In our view, the FOSS process can be described as the emergent behaviour of a large number of parallel iterations, varying but short in size. Nearly all FOSS activity consists of this kind of parallel iteration, whether it is design, code writing, testing, documentation writing, debugging, or something else. All the phases of more clear-cut models occur more or less at once, similarly to the Unified Software Development Process, but the iterations can be significantly shorter. It is not uncommon to have several iterations occur per day. In large projects with many participants, the iterations frequently become shorter and more numerous, while smaller projects progress at a slower pace.

Each individual iteration includes only a small subset of the project participants, often just one or two developers, while the others are observers to that iteration. This is supported by the tools that FOSS has developed for its own work. Mailing lists and on-line chat are used for the inception phase of each micro-iteration. Collaborative on-line editing tools, such as the *wiki*, are used to create documentation and plans. Recent version control tools support completely distributed work and allow each individual developer to handle a complete, separate branch of the entire project source code – the work process ensures that changes are shared and applied efficiently. These findings are similar to those of Huntley, who observes that while the individual developer's actions may appear chaotic, the overall learning process is fairly rigorous with specialised tools to support each stage [Hun03].

Huntley describes the FOSS method as a continuous organisational learning process, "where the development starts with a rudimentary but useful implementation, which is then iteratively improved as the team learns about use cases, designs, and coding techniques through experiment and user feedback" [Hun03]. However, not all projects are learning efforts, as Nakakoji et al. demonstrated [Nak02].

We note that the previous observations correspond remarkably well with the insights presented by Royce in his description of large software system development [Roy87]. It is ironic that the most valuable part of his contribution, the need for flexibility in the process, has been carried forward by a working method that has generally eschewed process as a concept.

When observing FOSS work, the overall process can be impossible to distinguish, because participants are constantly adapting and optimising for current requirements. Therefore, in Michlmayr's definition of process, what is described is actually quite technical and may be characterised as the working methods of the project rather than a high-level, managerial process. Nevertheless, to apply the insights of Schneidewind, understanding that FOSS views process very differently than traditional software engineering is vital when asking both *when* and *why* to measure quality [Sch02]. We propose that quality in FOSS projects should be measured at the level of each micro-iteration as well as by an overall, separate quality assurance process.

3.1.2 Process maturity and success

Process maturity is often cited as a prerequisite for repeatable quality. Like Polančič et al., Michlmayr hypothesises that there is a link between process maturity and the success of a FOSS project [Pol04, Mic05c]. Michlmayr observes that FOSS projects have an important volunteer component among contributors, and that they are usually globally distributed. This leads to a different set of motivations and a different value system than that of traditional software organisations, and a re-evaluation of traditional software engineering insights in a FOSS context is required.

Michlmayr uses the amount of software downloads as an indicator of success. It can thus be inferred that this view of success relates to popularity, and that popularity can be an indirect indicator of quality. However, a large part of available FOSS projects were started as mere experiments or as hobby activities, where success is not necessarily defined as popularity, or at all.

Michlmayr's methodology does correctly observe that comparison can only be meaningful if the compared projects are similar in all aspects except those whose relation is under scrutiny. He compares successful projects with unsuccessful ones and attempts to detect whether the process maturity differs in the two sets. Michlmayr finds that the maturity of the process employed in FOSS projects has an impact on the success of the project. However, the nature of this relationship is not investigated and Michlmayr suggests thorough code quality comparison and qualitative evaluation of release strategies as further research.

Crowston et al. have examined the work practices – effectively, the development process – of FOSS projects from the perspective of organisational theory [Cro04]. They describe a hypothetical, layered team structure in FOSS projects. Mockus et al. have made similar findings in their study of Apache and Mozilla [Moc02].

The team structure reflects the role of project participants (Figure 3). At the centre are the core developers, including the project founder and release coordinator(s). The next layer consists of co-developers who contribute more sporadically. In the following layer are the active users, who participate in development by testing new versions of the software and submitting bug reports and feature requests. Those are followed by readers, who may look at the source code, compile it for their own use, and make local changes, but do not communicate their findings, or only participate in discussions very sporadically. The outermost layer, whose size is most difficult to estimate, is that of the passive users who only use the software.

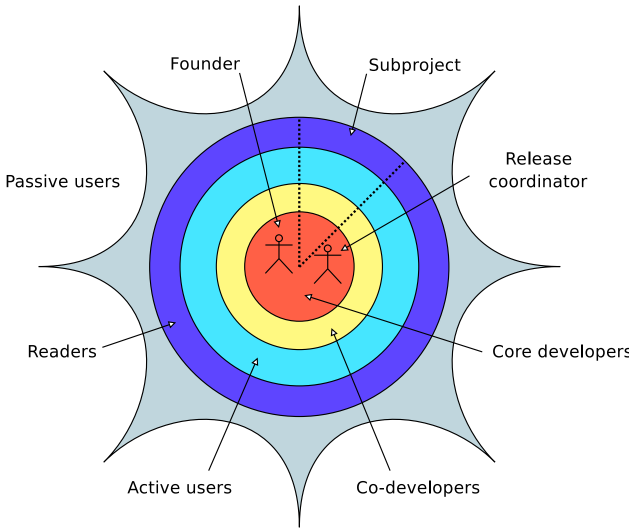


Figure 3: Layered organisational structure of an idealised FOSS project. Adapted from several sources and observations.

As Mockus et al. note, this structure has not been verified. It represents a simplified, prototypical model of a FOSS community. Its value lies in the insight that FOSS projects are not anarchic; they have intricate team dynamics, power structures and work processes that enable or prevent quality assurance.

Crowston and Howison observe that membership in inner layers changes less frequently than in the outer ones, and change at the core layer may be very infrequent [CrH06]. However, successful change can be a sign of maturity, as can the current composition of the inner layers. Therefore, examining the organisational structure of a FOSS project is important background information in quality assessment. Anomalies in the organisation may pose a risk to the continuity of the entire project, or result in neglected quality assurance.

3.1.3 Openness and reliability

Van Wendel de Joode and de Bruijne have explored the relationship between openness and reliability [WeB06]. The authors conducted a small survey to explore the relationship between source code access and defect correction. Respondents agreed that there is a relationship, but when asked to give further explanation, they concluded that "most users do not analyse the source code or try to report and solve bugs", that "half of the bugs are resolved by only one person" and "the ten most active bug reporters are responsible for 85% of all bug reports". The authors conclude that access does not guarantee quality.

Using organisational theories, the authors attempt to gain some insight into the relationship between openness and reliability. One theory, *normal accident theory*, states in part that if the elite in a group is unaffected by quality failure, then resources will not be allocated toward ensuring safety and reliability. It also states that unencumbered access to information within the organisation is vital, as it allows the organisation to learn from mistakes. Another theory, *high reliability theory*, states that certain organisations have nurtured conditions where reliability can be maintained despite the complexity of the systems managed. A strong presence of external, stake-holding observers is one of these conditions.

Drawing from these theories, van Wendel de Joode and de Bruijne formulate three hypotheses on openness and reliability in FOSS communities. First, corresponding to the elite: the bigger the percentage of developers who actually use the software they develop, the more reliable the software. Second, corresponding to organisational learning: the more transparent the flow of information, the more reliable the software. Third, corresponding to external observers: the more popular the software, the more reliable it is [WeB06]. These hypotheses, while certainly not empirically proven, correspond well with what can be seen in real FOSS projects.

3.2 Previous work and case studies

A quick glance at some random FOSS projects may lead to conclusions that there is no planning, requirements, quality goals, or project costs involved. Deeper knowledge of several FOSS projects reveals that thought is given to many complex quality-related questions, and that significant amounts of time and money have been invested in FOSS. However, many projects have not explicitly taken all the possible steps to correct their quality issues.

Furthermore, obtaining samples from FOSS projects requires deep understanding of the tools with which FOSS development is carried out. Enormous amounts of data is generated and stored in a wide variety of formats and in places that are not found without some technical knowledge. For instance, while some research claims it is impossible to know the number of developers involved in a particular FOSS project, this information is actually available to a great extent, for example in the package database of the Debian GNU/Linux distribution [Rob06] and in files accompanying the source code. It is important to review existing FOSS quality research and study FOSS projects in detail to gain an understanding of what information is available.

3.2.1 Growth, evolution and structural change

Godfrey and Tu have written a series of papers concerning the evolution of FOSS. They employ mostly statistical measures to examine the well-known Linux kernel project from a source code perspective. They also draw on observations of FOSS development principles and philosophy in general, and combine them with the measurements to explain certain behaviour in the Linux project [GoT00, GoT01].

Lehman's laws of software evolution state, among other things, that systems growth leads to increasing difficulty in adding new code unless explicit steps are taken to reorganize the overall design [Leh97]. However, Godfrey and Tu find that the evolution of the Linux kernel has seen a dramatic increase in size over a period of six years, contradicting Lehman's hypothesis [GoT00]. The authors attribute this to a number of factors, the most important being that the Linux kernel source code consists mostly of device driver code, and that the actual core is relatively small.

Further, Godfrey and Tu observe related work indicating that a top-level, full view of the modules or subsystems of a large code base is not enough to detect the important evolutionary behaviour of the software process, and that a more detailed view of individual parts is needed. In their own work, it was important to detect the distinction between the core code of Linux, and the relatively large body of device driver code. When viewed separately, different conclusions could be drawn about the kernel compared to an overall view.

The authors compared the overall size of the source code and the size of the different subsystem modules in different versions of the Linux kernel over time. In other words, they tracked the evolution of different branches of the kernel, and the evolution of the different subsystems. They also used external knowledge of features and changes that the kernel developers had introduced at different points in time, such as the porting of Linux to several different processor architectures. The authors did not expand their research beyond LOC analysis.

Although Godfrey and Tu present statistical evidence that seems to contradict Lehman's law, they fail to apply their own conclusion about the need for a more detailed view to their own claim of contradiction. Since the Linux project is divided into several parallel projects, each maintaining a particular subsystem, the project as a whole does not have to work with the entire code base at once. It appears that Lehman's explicit steps *have* indeed been taken in the Linux project to organise the overall design of the system as well as that of the working process in a manner that supports the current size and complexity of the system. Godfrey and Tu may have become blinded by the surprise of the total LOC evolution curve. In any case, later studies have found that the super-linear growth has at least not been permanent.

Izurieta and Bieman have repeated some aspects of Godfrey's and Tu's study and also applied the analysis to the FreeBSD kernel [IzB06]. In particular, they found that the claim of super-linear growth in Linux had no support. Both Linux and FreeBSD displayed sub-linear or, at most, linear growth, which is what previous studies by Lehman on industrially produced software have shown [Leh97].

There are several important methodological aspects of Izurieta's and Bieman's study. They used the existing project directory structure to divide the project into modules for individual study, as suggested by Godfrey and Tu, to eliminate observer bias. They used simply physical LOC, and measured program size by LOC. They also provided an interesting overview of the different released versions of FreeBSD. Their study concentrates only on the released versions of the software, and thus fails to capture the impact of ongoing work.

3.2.2 Mining public repositories

Since our goal is to automate the collection and analysis of metric data as far as possible, we must consider the practical possibilities and difficulty of data mining in publicly accessible data repositories. Koch and Schneider have presented a methodology similar to that of Godfrey and Tu. Their metrics and analysis displays a wider variety, taking into account the continuous development process that takes place in the source code repository and project discussion lists. They apply their methodology to the GNOME project [KoS00].

Koch and Schneider use a more rigorous approach than Godfrey and Tu, first gaining an understanding of the relationships between the different pieces of data by modelling them using an entity-relationship diagram. The model encompasses the people involved in the process, the discussion guiding it, and the actual source code produced. The model is based on observation of discussion list activity and activity within the GNOME CVS source code repository.

Koch's and Schneider's methodology displays important characteristics that allows capturing a number of defining aspects of FOSS development. First, the sample granularity is much smaller than in Godfrey and Tu. For source code, the sample is a single CVS commit action, instead of an entire release. Second, each action is associated with the person who performs it. Godfrey and Tu have no concept of *actor* in their model, whereas Koch and Schneider demonstrate that it is an important factor in the FOSS project. Third, Koch and Schneider include the discussion lists in the model, and although they do not analyse the discussion contents, they do find temporal correlation between discussion list activity and CVS activity. Fourth, Koch and Schneider apply time-line analysis throughout their methodology. Each action is placed on a time-line with very fine granularity, allowing the authors to make effort estimations and track progress on a very accurate scale, among other things.

In this work, LOC remains the primary metric to describe the size of source code and the size of changes to source code. The authors describe the different metrics where LOC is used: LOC added, LOC deleted and LOC changed, the last of which is defined as the difference between the first and the second during a given time period. The authors use a simple definition of LOC, counting physical lines regardless of their contents.

For the discussion lists, the authors define the metric *number of postings*. The authors then go on to define the *checkin* metric, and finally proceed to define derived metrics such as *time spent on the project*, *programmer activity*, LOC *added per checkin* and LOC *added per hour*.

Using these metrics and the correlations of the data model, the authors make a number of observations. They observe that the contributions to source code for each programmer follows a power law: a majority of programmers contribute only a small amount of code, while the bulk is committed by a small core team. Further, they attempt to define certain correlation patterns that concern the amount of change in the code, the tendency of active programmers to use smaller or larger commit chunks, the difference in programming style and the connection between long project membership and amount of contribution.

The authors observe similar patterns in the discussion lists: a small number of programmers contribute to most discussion list postings, while the majority of posters contribute quite infrequently. Correlating postings with LOC added, the authors conclude that programmers who contribute more code are also more active in the discussion lists. However, the authors could not find a correlation between the total amount of postings and the total amount of added code within a given time interval.

The authors derive a number of interesting trend graphs. For example, they attempt to detect whether a particular module of the GNOME project is undergoing an acceleration in development, or if it is nearing completion and development is slowing down.

The authors confirm that a higher number of contributors lead to more output. However, they note that the organisational structures may not support more than a given number of participants. When this saturation point is reached, an organisational change is needed to sustain output growth, an observation that is consistent with the findings presented by Mockus et al. and Dinh-Trong and Bieman [Moc02, DiB04, DiB05].

Asklund and Bendix have successfully captured the state of the art in FOSS configuration management in 2002 [AsB02]. They note that traditionally, configuration management has been a manager activity, where the stages of the software development process have their defined activities within a configuration management tool. These activities include configuration identification, where the product structure is determined; configuration control, where changes to a configuration item is performed; configuration status accounting, where the status of each change is recorded and deviations from the specified basic configurations is noted; and configuration audits, where configuration items are checked for conformance with their configuration, such as performance characteristics.

In FOSS however, configuration management is a developer activity, and includes version control, build management, configuration selection, workspace management, concurrency control, change management and release management. These are so central to FOSS development that it is only in the most special cases that they are not used in some form. While Asklund and Bendix make several assumptions and conclusions that are no longer valid in a modern FOSS environment, they have correctly identified configuration management as a crucial enabler for the distributed work method common in FOSS. The important activity happens within the source code repository, which is an increasingly distributed environment of which each developer holds a copy. To properly understand the decisions behind the source code changes, researchers must correlate mailing list and other on-line discussion activity with actual code commits. Asklund and Bendix correctly emphasize the importance of open communication to coordinate and plan the programming.

Massey reports challenges when obtaining data from FOSS version control systems [Mas05]. Extraction of data seems simple, but in practice it is not. A particular problem is that FOSS projects have periodically restarted their version control, either due to problems in the version control software, because of project forks, or because the project has moved to a newer version control system. However, we note that it is not always necessary to obtain or analyse the entire version control history. Instead, intelligently selecting a certain period of time can be sufficient and useful to explain important events in the history of a project, as noted by Massey. Furthermore, analysing and recording the current activity could yield usable results in a relatively short time. Models based on historical data may be invalid if the project has changed its production process.

3.2.3 Empirical tests of statistical quality models

Zhou and Davis have empirically tested a software reliability model on eight FOSS projects [ZhD05]. They confirm that the defect arrival rate follows a Weibull distribution, but find no support that it would follow the more specific Rayleigh distribution commonly found by the industry in non-FOSS software. They suggest that each FOSS project may require its own Weibull shape parameter, and that the parameter may change over time depending on several factors. They conclude that the defect arrival rate stabilises at a low level if the project continues for long enough. Similar results are reported by Tamura and Yamada, who emphasise that interaction between software components should be taken into account [TaY05]. Also similar is the stabilisation time metric proposed by Sharma and Jalote, as described in Section 2.3.4 [ShJ06].

The significance and utility of these findings is that stabilisation of the defect arrival rate has been used in the industry as a sign of reliability. When the curve stabilises, testing can cease and the software can be released. Adapting this to a FOSS environment could be an important tool for measuring overall quality and aid in release management.

Li et al. also attempt to fit a Weibull model to development defect arrival, examining the OpenBSD project [Li05]. Their results show that it is not possible to obtain results before most of the defects have occurred, meaning that there is insufficient data to fit the model. Thus, they conclude that this approach is infeasible, and turn to metrics-based field defect prediction instead. They find the most important predictor to be the number of messages to the technical mailing list during the development period. The lack of causal information suggests that this may be a case of confusing cause and effect – it is not the preceding discussion that creates the defects; it is the testing and search for the defect that causes the discussion, which in turn causes the defect report to be filed. Nevertheless, the increase in discussion may indicate that the project does perform active quality assurance.

Phadke and Allen have made similar experiments, and conclude that quality modelling is viable for predicting modules with a high risk of defects [PhA05]. They compare logistic regression and decision trees, and while both methods do perform correct classification in some cases, neither produce satisfactory results over their entire data set.

Dick and Sadia argue that parametric statistical models are not suitable for accurate quality prediction [DiS06]. As there is no conclusive theory that maps software metrics to software defect rates, research in this area is reduced to a trial-and-error search through an infinite space of models. Instead, useful results could be achieved by using machine learning. At the very minimum, managers could benefit from a binary decision about the expected quality of a code unit – that it needs or does not need additional effort. They also note that the lack of large data sets with many software metrics and defect data included is problematic, but in FOSS projects, these items are available.

The authors further note that no quality assessment process or metrics currently exist for FOSS, but they do not state why or how it would be different from non-FOSS. They collect a data set from the Mozilla project, and then analyse it using fuzzy clustering algorithms. They conclude that their method was able to identify groups of modules with a higher defect density than traditional models, yielding a basis for a useful tool in quality assurance.

3.2.4 Object-oriented metrics for FOSS

Gyimóthy et al. have validated a number of object-oriented metrics for fault prediction on FOSS projects [Gyi05]. The analyses used to validate the metrics were regression analysis, logistic regression, and machine learning. All methods yielded similar results and thus they should have high validity according to the authors. They were able to validate the WMC, DIT, RFC, CBO, LCOM, LCOMN, and LOC metrics as valid predictors of future faults. The best single predictor was CBO, but LOC also performed well. DIT and NOC metrics were unsuitable for fault prediction.

Studying object-oriented programs is not without complication. Some of the object-oriented languages widely in use, such as C++ and Java, provide some features of generic programming. Briefly, this allows the creation of classes that are generic with respect to the objects they can interact with. The compiler, preprocessor or virtual machine will instantiate versions of these template classes for each class that the template class will interact with. This poses a problem for analysing code, because the class hierarchy, and thus the program architecture, is not the same when analysing the source code and the code that is actually run. Also, program architecture may be very hard to recreate without duplicating the operation of the build system.

Gyimóthy et al. used a data-collecting script as a wrapper for the build tools and were thus able to extract information from source code that was generated during the build. Compiler wrapping was used previously by the same authors to detect fault-proneness in the Mozilla suite [Fer04]. The authors describe the technique and their overall fact extraction process in detail.

The process was carried out using Columbus, a reverse engineering framework, and consists of five steps (Figure 4). The first step, acquiring project and configuration information, is particularly problematic since the automatic actions of the build system might produce code that should be analysed, but escapes analysis because it is only available during the build. This is in contrast to Robles et al., who explicitly avoid counting automatically generated code, presumably because of the tool they used [Rob06].

Ferenc et al. redirected calls normally made to the compiler and other build programs to a wrapper script. This script behaved externally exactly like the original program, accepting the same input and parameters and producing the same output and return value. It invoked the real program with the original parameters, but it

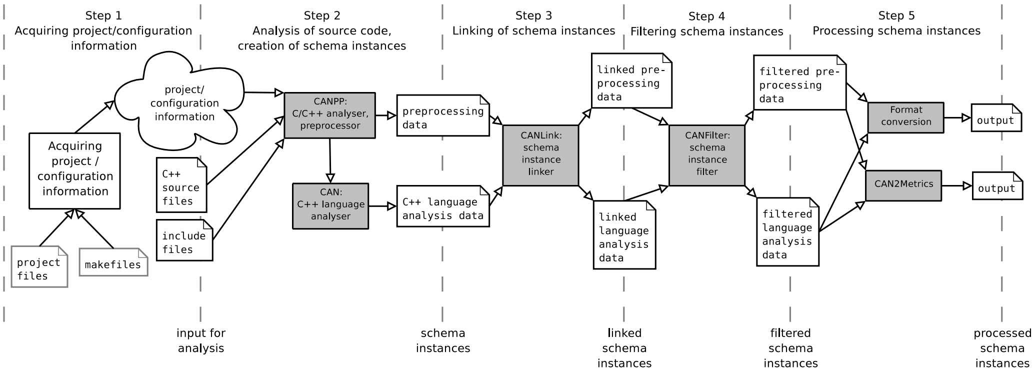


Figure 4: Fact extraction process of Columbus framework [Fer04].

also invoked another program that arranged the analysis of the output files, allowing the automatically generated code to be analysed as well.

Ferenc et al. analysed Mozilla using the previously described technique and a number of hypotheses about object-oriented metrics (Table 4). They did not attempt to produce any new validation of the metrics, and relied rather on previous results. They found decreased overall fault-proneness in Mozilla from version 1.0 to version 1.6, but did find individual classes with increased fault-proneness.

Ferenc et al. justify their choice to examine the compile-time architecture by observing that the source code is split among several different files and directories, and that the architecture must be extracted before proper analysis can be made. Another view is that the generated code is not written by humans, and thus represents no

Metric	**Hypothesis**
WMC	A class with significantly more member functions than its peers is more complex and, by consequence, tends to be more fault-prone.
DIT	A class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors.
RFC	Classes with larger response sets implement more complex functionalities and are therefore more fault-prone.
NOC	Classes with a large number of children are expected to be more fault-prone.
LCOM	Classes with low cohesion among its methods suggests an inappropriate design which is likely to be more fault-prone.
CBO	Highly coupled classes are more fault-prone than weakly coupled classes.

Table 4: Object-oriented metric hypotheses used by Ferenc et al. [Fer04], originally drawn up by Basili et al.

increase in programming effort. Also, one could argue that if one layer of abstraction is removed, the next layer should also be removed by the same reasoning. This would lead to analysis of the compiled object form of the program. If the metric is to capture some aspect of human effort, it is not advisable to measure automatically generated code. If the architecture can be extracted during the build, it must be possible to recreate it from the source code, since this is what the build system does. However, extending the build tools, including the compiler, to output metrics at build-time is more practical than writing completely new tools that duplicate already implemented functionality.

3.2.5 FOSS quality assessment model

Polančič et al. describe a software selection model specific to FOSS projects that uses easily accessible data [Pol04]. The model is based on a manual procedure. The authors observe that the model must not be too time-consuming or expensive to use, because this is impractical in a FOSS setting. The proposed model consists of six steps. First, software alternatives are identified. Second, quality metrics are identified based on the amount of information available for the alternatives. Third, metric weights are assigned on a scale from one to three. The authors recommend that multiple experts select the weights to reduce subjectivity. Fourth, metrics are calculated and normalised. Fifth, quality characteristics values are calculated and normalised. Finally, weights are assigned to quality characteristics by the party requesting the evaluation.

The method is quite straightforward and thus fulfils the requirement of simplicity, but there are some issues. Some steps are very subjective, although this could be ignored when the same experts review several software alternatives at once for the same client. However, the model is very loose when defining quality metrics, and none of the quality metrics used in the authors' experiment measure the actual source code of the program. The model seems to be focused on measuring FOSS project meta-data, but does not validate its quality impact.

The utility of the method is not clear. Although the authors claim that it could be used by a FOSS user when choosing among software alternatives, it ignores several aspects. The potential of young projects is ignored because the age of the project is directly correlated with functionality, reliability and usability without any consideration of actual work. The willingness of the user to participate in development or funding of the project, or the time frame within which the user wishes to take the software into use are also relevant to such selection. Project meta-data might not be usable at all except for comparing the project's own view of itself against a more objective analysis.

4 Quality model for Free and Open Source Software projects

There are many ways in which the quality of FOSS may be assessed and improved. Software consultants in the industry use simple methods, similar to the one described by Polančič et al., to assess the quality of FOSS products before recommending them to customers [Pol04]. Integrators test the software using both static and dynamic methods before and during deployment [MäM06]. These practices are likely to be very specialised and the results are seldom reported back to FOSS projects. Quality assurance that can occur both within the FOSS project itself, as well as externally, provides an opportunity for quality improvement instead of mere acceptance or rejection of the software as a whole. At best, the project incorporates continuous measurement and, as a result, grows more mature.

We shall focus on metrics as a primary means of obtaining information, and our base assumption is that certain process traits are key to continuous achievement of quality. Our aim is not to describe a quality assurance process for some particular product, but rather to explore how quality factors in FOSS projects can be measured and tracked generally. Our model is generalised to fit the most common traits in most FOSS projects. Applications will reduce or adapt the model according to each particular case. Given a list of real requirements, this approach can be applied to measure the quality in a product for a particular user.

4.1 Preliminary description and project hypotheses

Wake and Henry have observed several quality studies and note that it is important to examine changes in software from one version to the next, because maintenance activity results in structural complexity growth, a quality risk [WaH88]. They are inspired by the "ripple effect", among other individual metrics, and attempt to correlate different metrics and quality observations.

In their experiment, Wake and Henry use a "code library" – probably an equivalent of a source code management or version control system – to track changes to the studied software over time. The smallest unit of change in this system is a single line of code. Additions as well as deletions of lines are stored, and a modification is identified when a deleted line is followed directly by an added line.

Wake and Henry describe a number of additional facts that can be obtained from the code library. An important one is in which procedure each change has occurred. The authors note that by tabulating the additions, deletions, and number of changes for each procedure, they obtain an indication of the maintenance activity that the program has gone trough during a specified period of time. This is an important observation for FOSS, where version control tools are frequently used. Wake and Henry use these facts as control data against which they verify the experiment.

The goal of the experiment was to predict how many lines of code were changed during the maintenance phase, based on a number of metrics. In other words, change in lines of code is the dependent variable and the metric values are independent variables in the statistical model. Wake and Henry chose to use multiple metrics because in initial tests with single metrics, the model was not accurate enough. Better accuracy was achieved using a multiple regression model with several metrics, but they still note that the selection of the metrics pose a challenge.

The authors compared and ranked several models, and then selected one for presentation. They showed a prediction example where they consider two hypothetical procedures in the software. They calculate Halstead's effort metric, McCabe's cyclomatic complexity and a metric further developed from information flow, information flow with effort. The prediction equation directs the project manager toward one of the procedures where changes are likely to be required. Naturally, to act properly on the indication, a programmer needs to understand the code and make the proper change. The method does not say what is wrong with the code.

However, version control systems include only the source code of the product. There are several other sources of information. Mockus et al. find that mailing lists, version control systems and bug tracking systems include important information about FOSS projects [Moc02]. They pose seven hypotheses about FOSS projects, obtained by first observing the Apache project and then refining their findings by observing the Mozilla project. Dinh-Trong and Bieman have checked the hypotheses against FreeBSD, and found support for some of them [DiB04, DiB05].

The first hypothesis is that FOSS projects will have a core of developers who control the code base, and will create approximately 80% or more of the new functionality. If this core group uses only informal means of coordinating their work, the group will be no larger than 10 to 15 people.

The second hypothesis is that if a project is so large that more than 10 to 15 people are required to complete 80% of the code in the desired time frame, then other mechanisms than informal arrangements will be required to coordinate the work. These mechanisms may include explicit development processes, individual or group code ownership, and required inspections.

The third hypothesis is that in successful FOSS projects, a group larger by an order of magnitude than the core will repair defects, and a group larger by yet another order of magnitude will report problems. Taken together, these first three hypotheses imply that project structures will break down if the group grows large, and there is no effort of coordination. The composition of human resources in a FOSS project is a quality factor.

The fourth hypothesis concerns projects that have a strong core of developers but never achieve large numbers of contributors beyond that core. It states that they will be able to create new functionality, but will ultimately fail to become sustainable because their resources do not suffice to find and repair defects.

The fifth hypothesis is that defect density in FOSS releases will generally be lower than in commercial code that has only been feature-tested, that is, received a comparable level of testing. The sixth hypothesis is that in successful FOSS projects, developers will also be users of the software. The seventh hypothesis is that FOSS projects exhibit very rapid responses to customer problems.

4.2 Data sources

It is easy to fall into the trap of only analysing data that is easy to obtain, such as automatically collected download statistics, or statistics produced by FOSS projects themselves. This may skew results as the easily obtainable data may highlight only a single viewpoint. Similarly distorting is the practice of disconnecting artefacts and the people who produce them. FOSS is an activity where discussion and code interact and are mixed. Discussion can frequently occur in or through the code, and vice versa [Bar05].

To avoid distortion and one-sided analysis, we include data sources from three principal information spaces (Table 5). In the *discussion space*, project participants discuss and negotiate different aspects of the project using discussion lists and fora, as well as other means of communication. In the *documentation space*, the project documents both its rules and practices and creates user documentation. Documentation external to the project exists in this space. In the *implementation space*, the actual software deliverable is produced, often using a network-accessible code repository or versioning system. Also, the implementation space is where users of the software deploy the project deliverable. A similar taxonomy has been used in other studies [Bar05]. We will subsequently describe the three spaces in greater detail.

4.2.1 Discussion space sources

The discussion space of a project consists of mailing lists, web fora, newsgroups and other real-time and non-real-time communications systems. In the discussion space, participants contribute to a free flow of information. Individual participants can be identified by their email address, name, or another identification token. Participation frequency, participation volume, the number of discussion threads the participant takes part in, and several other metrics can be calculated. It is possible to connect the actions of a participant to actual work in the documentation or implementation space, allowing observers to explore the relationships between the three spaces.

In the discussion space, the flow of information is carried forward using several methods. Barcellini et al. have compared a threading model and a quotation-based model and concluded that the latter is better suited to analyse design-oriented discussions [Bar05]. These discussions can be evaluated to assess design quality. However, it is computationally intensive and hard to automate. A simpler alternative is to measure only the frequency and volume of such discussions. This provides some insight into how much design is performed overall.

4.2.2 Documentation space sources

In the documentation space, contributors can be identified for each artefact. As project-level documentation is revised over time, some contributors will have the status of initial author, while others will have an editorial role. This can be correlated to participation in the discussion space and implementation space.

Information space	Data category	Data sources
Discussion	Real-time	IRC and other on-line chat networks
		Telephone, teleconference, including VoIP
		Face-to-face discussion and meetings
	Non-real-time	Mailing lists
		Web fora
		Newsgroups
		Personal email
		Blogs
Documentation	Project goals, mission and vision statements	Web site
		Wiki, mailing lists, web fora
	Administrative documentation	Web site
		Wiki, mailing lists, web fora
	Development documentation; coding style; granularity of changes; branch usage; commit policy	Web site
		Wiki
		Mailing lists, web fora
		Documentation in or with source code
	User documentation: manuals, guides, tutorials, etc.	Web site
		Wiki
		Documentation in or with source code
	External documentation	External sources, e.g. third-party web sites
Implementation	Accepted code	Version control system
	Proposed code (patches)	Bug tracking systems
		Mailing lists, newsgroups, web fora
	Defect reports	Bug tracking systems
		Mailing lists, newsgroups, web fora
	Feature requests	Bug tracking systems
		Mailing lists, newsgroups, web fora

Table 5: FOSS information spaces and data sources. The reliability of each source must be evaluated separately. This list of data sources is not conclusive, as new tools emerge continuously.

Furthermore, the availability and type of project-level and development documentation can affect the sustainability and quality of the project. For example, complete lack of a coding style decision could lead to inconsistent code, which then becomes difficult to maintain over time as new developers enter the project. On the other hand, lack of written documentation could be compensated in the discussion space, as new contributors learn how the project works by direct feedback or by looking at previously given feedback. In that sense, some of the discussion space artefacts may transfer into the documentation space. Conformance to existing documentation can be assessed by comparison with the actual implementation. If the documentation is not followed, this could be due to lack of agreement or understanding of the documentation, or due to outdated documentation.

The existence of user documentation must be viewed in light of the project goals. Not all programs need extensive documentation, but if it is directed to non-technical users, documentation can be an essential component.

4.2.3 Implementation space sources

Perhaps the most important data sources are the implementation space sources. They form the main part of the project deliverable, and most of the work is assumed to be directed to them. Since the implementation space artefacts are often stored in a source code repository with versioning capabilities, extracting data over given time periods is possible, enabling research into software evolution. A change in quality over time can indicate the direction of the project, and allow the project to take corrective steps if quality is declining.

Another implementation space data source are the bug tracking systems that the project uses to keep track of defects and feature requests. Bug tracking systems are frequently used in FOSS projects to determine when the software can be released. A common development pattern is to work on a development version of the code and then go into "freeze", introducing no new functionality while bugs are being fixed. When certain criteria are met in the bug tracking system, a release can be made. Recently, attention has been given to time-based releases, where the release date is set in advance and a new version of the software is released if it meets quality criteria on that date [Mic05b, Mic07]. This method is most relevant when applied to software collections – the collection follows a time-based release schedule, and the parts which are ready by release time are released, while those that are not have old versions re-released.

When coupled with the discussion and documentation space sources, the background and reasons for changes in the implementation space can be evaluated, making it possible to assess how the three spaces interact and thus how the project functions as a whole.

4.3 Data model

Several systems to support metrics extraction and analysis, both theoretical and practical, have been proposed. Examples include the ATHENA software quality and certification tool [Chr89], Toshiba's ESQUT quality evaluation tool [Oga96], the Columbus tool by the University of Szeged, Nokia and FrontEndART [Fer04] and the SOCCER software quality evolution tool [Bou06]. Each of these have a slightly different focus, some emphasising a theoretical model and some solving very specific data-collection issues. However, no such tool has gained widespread use in the FOSS community.

In order to facilitate data collection and arrange the available data in the three information spaces, we have constructed a data model (Figure 5). It consists of the following classes. The *project* class represents a FOSS project. A project has a *web site*, a *version control system*, a *bug tracking system* and a *discussion forum*, but each of these can be absent or exist in multiplicity.

A web site consists of *pages*, authored by one or more persons. A version control system contains a number of *change sets*, consisting of *files*. Each change set is *committed* into the system by one *person*. A bug tracking system consists of *bugs*, whose modification over time is described in a log of *bug events*. Each bug event results from the actions of one person.

A discussion forum is a generalisation of *mailing lists*, *web fora*, and *news groups*. It consists of *posts*, each of which is created by a single person. Posts may be placed in one or several discussion fora, and may be related to other posts through a *followup* mechanism.

A person is a named entity, and the model can identify any number of *identifiers* that act as unique tokens to identify the user in the systems represented by the other classes. Furthermore, each person may possess one or more *blog*, consisting of *blog posts*. The posts may be in reference to other blog posts through a mechanism called *linkback*.

Although the model does not include all data sources presented, it allows the implementation of a system to collect, store and analyse metric data as it is created. The model does not show the details needed to support systems currently in use in FOSS projects. Actual data collection needs to consider several different access methods and data formats.

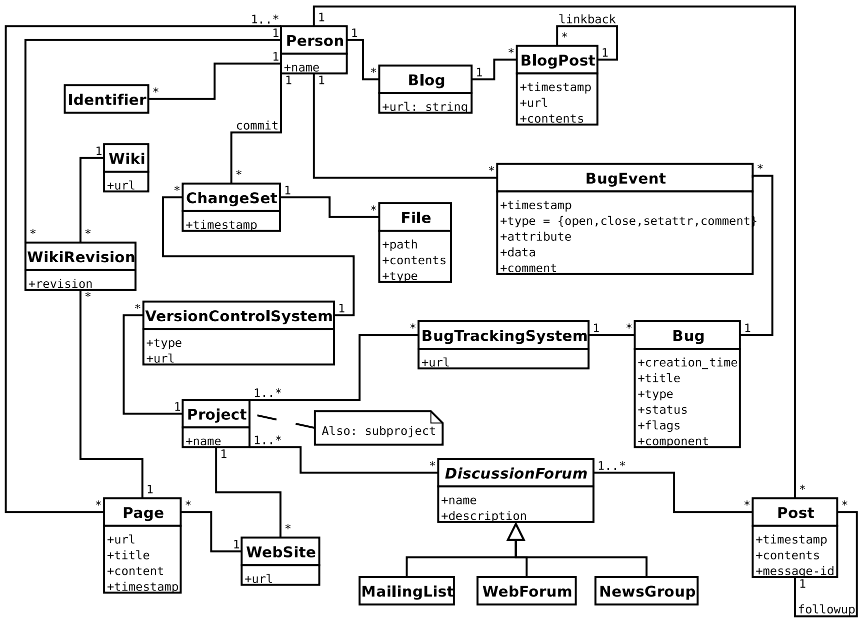


Figure 5: UML diagram of our FOSS project data model. Based in part on an ER diagram by Koch and Schneider [KoS00].

4.4 Metric taxonomy

Based on the research and observations presented so far, we construct our metric taxonomy (Figure 6). We make the following assumptions. Functionality is assessed by other means; projects have equal or comparable functionality, or they are the only available alternatives for some other reason. Cost is not considered, and is assumed to be constant or to be some function of quality. Legal issues, such as the software license, are not considered, and we assume that the projects to be assessed fall within an undefined set of minimum legal requirements for FOSS. The taxonomy is based on the IEEE SSQMM framework [IEE04].

The taxonomy divides the quality of a FOSS project into process quality and product quality. The process quality factors are organisation and work flow, and measures the most important traits that have been identified as crucial in the success of FOSS projects.

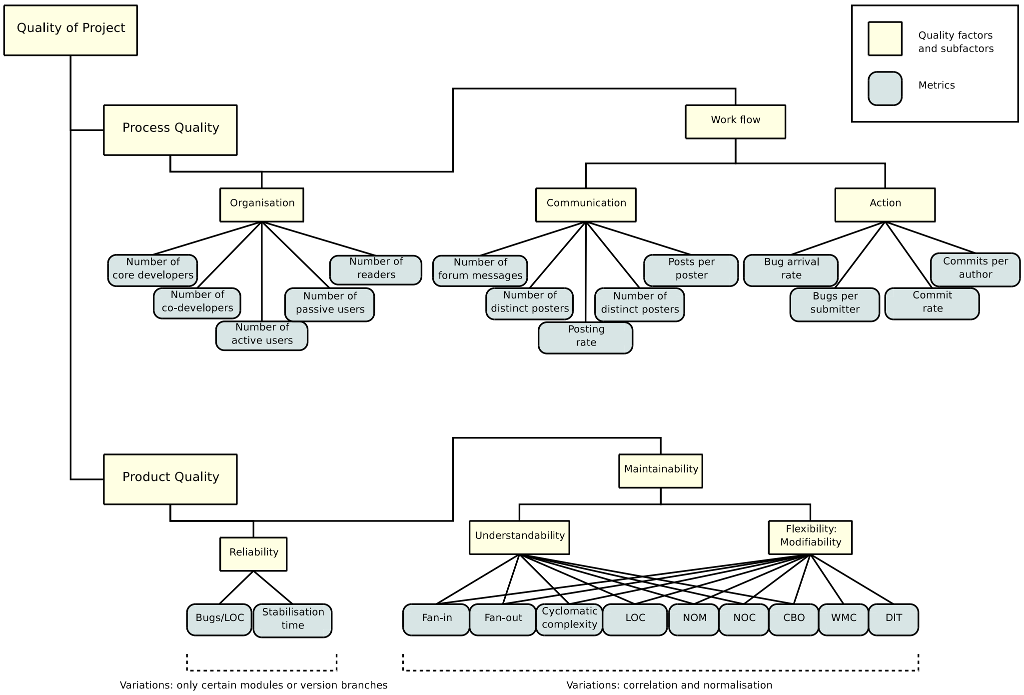


Figure 6: FOSS project quality model.

As product quality factors, we have chosen to reflect flexibility, fault density, and quality risk. Flexibility can be viewed as a form of maintainability measure, which estimates how easy it is to make changes to the program. Fault density is an estimate which balances two concepts: how good is the code that has been written, and how effective are the procedures we use for finding faults? Finally, quality risk estimates to what extent we have found the defects that exist or are only repairing peripheral symptoms, giving rise to new faults in the same area. Also, it measures whether the bugs found are concentrated to a particular part of the code or are spread evenly. The quality risk attempts to evaluate how large the potential is for defects to occur in the program or in a particular part of it. In other words, it reflects reliability.

Since FOSS developers have a very varying degree of knowledge, we have chosen metrics that are as easy to understand as possible. Otherwise, project participants may reject the model or make incorrect decisions due to misunderstanding. Since FOSS development is rapid and frequently produces amounts of code that is impractical to analyse manually, the metrics have been chosen so that they can be automatically collected and analysed with reasonable effort and computational resources. Also, their implementation is straightforward.

The model could be further extended, for example by including some of Boehm's quality characteristics, but this would ultimately lead to metrics that cannot be collected automatically. For example, interoperability is a quality characteristic that is very difficult to test automatically. It should evaluate data exchange formats and other interfaces of the program that allow communication with components that are not part of the program itself. Usability is left out completely – it is such a large field of its own that we could not hope to cover enough of it in this work. Another area of quality which has been left out is that of performance, both in the sense of efficiency and scalability. It is doubtful whether these have any general meaning, although they can be of great importance in specific cases.

5 Experiment: Applying quality analysis to real Free and Open Source Software projects

We have analysed three FOSS projects using the quality model described in the previous section. We first conducted an experiment prototype, in which we made feasibility tests for obtaining data and taking measurements from it. We experimented with approximately ten different FOSS projects, including the Linux kernel project and a few software packages of the Debian GNU/Linux distribution. Our prototype used a database schema directly derived from our data model (Figure 5) with small technical and practical changes. We wrote simple tools to collect data from several different sources and calculate the metric values. We also evaluated more than a dozen metric tools, and found some partly suitable to our needs. However, we were unable to find tools to cover all our needs.

The prototype revealed that many tools currently in use in FOSS projects are either not capable of easily and automatically exporting their entire event history, or an impractical amount of work would have been required to develop tools for extracting the complete event history. For example, the Bugzilla bug tracking system can only export the current status of a bug. Although the event history is stored, it cannot be accessed in an unambiguous format that would be easily machine-readable unless direct access to the back-end database is obtained. Another example is the Debian bug tracking system, where complete bug logs are available, but the log format is a human-readable representation which does not lend itself well to automatic processing. Therefore, we elected to use arrival rates for this experiment instead of a full time-line event analysis.

5.1 Project selection

From the initial set of approximately ten FOSS projects, we selected three: Linux 2.6, an operating system kernel, The GIMP, a graphics program for creating and manipulating two-dimensional images, and Blender, a 3D content creation suite which includes modelling, animation, rendering and interactive 3D features. Additionally, we included source code from old versions of the Linux kernel, ranging approximately from February 2002 to April 2005, as a comparison to the current Linux kernel tree. We refer to this code as Linux-historical.

These projects were selected because they are large and complex in terms of source code, features, developer base and user base, and because their code has had the opportunity to reach a certain maturity during at least five years. Together, they cover a range of quite different approaches to and stages of FOSS development; the Linux kernel has an extremely rapid pace of development, The GIMP is undergoing major refactoring to modernise it, and Blender is a highly specialised tool with a highly specialised audience and developer base. Also, the necessary data for all three was available and obtainable with reasonable effort.

5.2 Data identification, acquisition and cleaning

For each project, we identified the relevant data sources (Table 6). This information was obtained by visiting the web sites of each project, and by reading the project documentation available there. This was trivial – less than an hour of time was spent on obtaining the information. Also, all three projects have an enormous mass of documentation available, ranging from user manuals and tutorials to complete books. They cover a wide range of audiences and are of many different levels of quality. Therefore, we decided not to include the web sites and documentation in this comparison.

To acquire the data from each data source, we wrote special programs based on the earlier prototypes. All programs insert the results of their computation into an SQL database. The values of each computation can then be retrieved efficiently along with information about its context – from which project it was obtained, to which point in time the value is connected, and so on. The programs were written in the Python language.

The first program processes compressed mailing list archives in the `mbox` format. The messages archives were downloaded manually. For each message, the program

	Linux 2.6	Linux-historical	Blender	The GIMP
VCS type	git	git	SVN	SVN
VCS URL	git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git	git://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git	https://svn.blender.org/svnroot/bf-blender/trunk/blender	http://svn.gnome.org/svn/gimp/trunk
BTS type	Bugzilla	Ad-hoc / none	GForge	Bugzilla
BTS URL	http://bugzilla.kernel.org	Various / none	http://projects.blender.org/tracker/?group_id=9&atid=125	http://bugzilla.gnome.org/
Relevant mailing lists	LKML	LKML	bf-committers, bf-python	gimp-developer
Mailing list archives	http://userweb.kernel.org/~akpm/lkml-mbox-archives/	http://userweb.kernel.org/~akpm/lkml-mbox-archives/	http://lists.blender.org/pipermail/bf-committers/, http://lists.blender.org/pipermail/bf-python/	https://lists.xcf.berkeley.edu/lists/gimp-developer/

Table 6: Data sources for each project.

extracts the sender, subject, message identifier and the date when the message was sent. These are subjected to data cleaning. For example, the mailing list archives for the Blender project had some of the sender addresses set to the mailing list address. In this case, we used only the sender name as identifier. In other messages, the sender address was obscured by replacing the `@` sign with the word "at". We reversed this in the data cleaning stage. Finally, the date of sending was specified in a wide variety of formats. We wrote logic to convert each of these into a format that could be used by the database. Unfortunately, the date in some messages was impossible to repair. As these cases were relatively rare, we decided to omit the messages from analysis. Table 7 shows the number of messages and the time required to import them into the database for each mailing list.

Mailing list	Messages imported	Time required
LKML	646 001	approx. 7 hours
bf-committers	19 029	approx. 30 minutes
bf-python	4862	approx. 10 minutes
gimp-developer	20 320	approx. 30 minutes

Table 7: Number of messages and time required to import them on a 1.2GHz PC.

The second program obtains bug reports from bug tracking systems via HTTP requests. We wrote support for the Bugzilla and GForge systems, since the first is used by Linux and GIMP, and the second by the Blender project. The program first obtains a list of all relevant bugs, and then downloads individual bug reports. Bugzilla was able to export the bug list in CSV format and the individual bugs as XML. We attempted to use the GForge SOAP interface, but were unsuccessful due to incompatibilities between the server and the Python SOAP client libraries we attempted to use. Instead, we parsed the GForge HTML pages by a series of regular expressions to extract the needed information. The number of queries needed was greater, since GForge splits the bug list into 25 bugs per page, and the submitter information is on a separate page.

Despite the fact that the number of queries to obtain data for n bugs were $q_{bugzilla} = 1 + n$ for the Bugzilla system, and $q_{gforge} = \frac{n}{25} + 2n$ for GForge, the latter was faster. The reason is not conclusive, but we note that our network connection to the Blender GForge server was better, and the parsing overhead of the Bugzilla XML data was significantly greater than the regular expression approach used for the GForge HTML data. Table 8 shows the number of bugs and the time required for importing them into the database.

Bug tracking system	**Bugs imported**	**Time required**
Linux Bugzilla	8954	approx. 8 hours
Blender GForge	4048	approx. 1,5 hours
The GIMP Bugzilla	7232	approx. 5,5 hours

Table 8: Number of bugs and time required to import them on a 1.2GHz PC.

The third program obtains source code from network-accessible repositories one revision at a time and runs metrics on the obtained source code with regular intervals. We wrote support for Subversion and Git systems, since the first is used by Blender and GIMP, and the second by the Linux project. Since these systems differ considerably in operation, our program abstracts the differences and reduces the operation of the systems to a common subset. The Subversion part first downloads the complete revision log, and then downloads the first revision of the repository. It can then step through the revisions forwards or backwards, and download the minimal set of changes, called a *diff*, to bring the local copy to the state of the specified revision. It can also download and show the diff between the current local copy and the state in which it was before the last operation.

The Git part works similarly, but the Git system supports distributed development. The source tree is branched every time a developer clones another developer's tree, and can be merged back either directly or via other developers. Several separate time-lines of development may exist simultaneously. This makes it very complicated and in some cases impossible to follow the global changes to the source code. We chose to observe development through one particular tree, the main tree maintained by Linus Torvalds. Changes made to other trees are reviewed and later included into this tree, temporarily merging the time-lines. Because of the way the Git system works, our program first downloads the entire tree of revisions, and then performs the required operations locally.

The performance characteristics of the two systems are different. The Subversion part is network-dependent in all stages, while the Git part is heavily network-dependent in the initial stage, and is then dependent on storage system I/O.

After obtaining the initial data, the program steps through each revision, noting the revision identifier of the underlying version control system and the author and date of the revision, and inserts these into the database. With regular, user-definable intervals, the program can run a number of metrics on the source code. These are further explained in Section 5.3. Table 9 shows the number of revisions imported, the metric run interval, and the time required to perform this data acquisition.

Source code repository	**Revisions imported**	**Full metrics interval**	**Time required**
Linux (git)	64 707	every 300 revisions	approx. 44 hours
Linux-historical (git)	63 428	every 3000 revisions	approx. 14 hours
Blender (SVN)	10 826	every 100 revisions	approx. 8 hours
The GIMP (SVN)	21 157	every 500 revisions	approx. 9.5 hours

Table 9: Number of revisions, run interval for full metrics calculation, and time required to import the data on a 1.2GHz PC.

All programs make use of the data already stored in the database to identify the actor whose action resulted in a mailing list message, a submitted bug, or a source code revision. As artefacts were analysed, the database was automatically consulted to see if the associated actor identifier had already been recorded. If so, we assume that the actor is the same person. Thus, we were able to make some correlation between actions in the different information spaces, although we did not use all the possible data sources.

On inspection of the discovered correlations, we found that the identification tokens have too little overlap between the different systems. In the Subversion system, the actor is identified only by a user name local to the main Subversion server. We found that this user name was frequently different from the name or email address used elsewhere, so unambiguously connecting the actors in all three data sources was not possible. We considered an approach where the analysis would be re-run each time a new identifier token is added, using tokens from previous runs to bootstrap the identification. However, the time required to run that many iterations would have exceeded reasonable limits.

Another approach considered was to manually produce the identifier tokens, but we rejected this because the time required would have been significant, and we wanted to see how well this completely automatic approach would work. We note now that the order of data source analysis is relevant; if we had started with the version control systems, we could have obtained greater overlap by using the local part of the actors' email addresses as well as the whole email address as an identifier in later stages. The identifiers obtained from the version control systems seem to be the most consistent and have the least amount of errors and variations.

5.3 Description of metrics calculation

As noted in the previous section, our programs calculated a set of metrics at regular revision intervals during the source code analysis phase. We divide the metrics into six families: CCCC metrics, static C metrics, static Python metrics, diff statistics, file type analysis and version control log analysis.

The CCCC metrics are produced by the CCCC program, a metrics tool for C, C++ and Java source code. These metrics were run against Blender and The GIMP. We were unable to run them against the Linux source code, because the program locked up during analysis of that code. CCCC produces a large amount of metric data, but we extracted only a number of metrics.

For the entire source tree, we extracted source lines of code, comment lines, lines rejected by CCCC, the cyclomatic complexity number, information flow, and the number of modules, for all files whose names ended in `.c`, `.h` or `.cc`. For each module, we extracted coupling between object classes, depth of inheritance tree, number of children, weighted methods per class, fan-in, fan-out, and information flow. We note that the definitions of these are given in the CCCC program. For example, the cyclomatic complexity is actually an approximation obtained by counting the number of decision point statements in the program, fan-in is the number of users a module has in its producer capacity, and fan-out is the number of modules a module makes use of in its consumer capacity.

The static C and Python metrics were written to extract simple syntactic structures from the source code. For the entire source tree, the metrics are source lines of code, physical lines of code, blank lines of code, comment lines, number of modules, an approximation of McCabe's cyclomatic complexity, obtained by counting the number of decision statements in the code, and variants of fan-in and fan-out that calculate the number of parameters in a function or method, and the number of return statements in a function or method. For individual modules, the same metrics were used and run against all programs for all files that ended in `.c`, `.h` or `.py`. We chose to include Python code, as it is used in two of our selected projects as a scripting and extension language, and since we have prior experience with automatic analysis of this language.

The diff statistics counts the number of files changed and the number of added and deleted lines of code. The file type analysis tries to guess the type of each file by looking at the file name. The version control log analysis extracts authors, time stamps and native revision identifiers from a version control system log. These were run against all programs. Table 10 shows all metrics and their codes.

For each program to be analysed, the full metrics were run against only a subset of the source code revisions. For the Blender project, full analysis was performed every 100 revisions, for GIMP, every 500 revisions, for the historical Linux code, every 3000 revisions, and for the Linux 2.6 code, every 300 revisions (Table 9). These numbers were chosen based on the observed activity in the project and practical considerations such as the time available to perform data acquisition and analysis. This provides a sufficient approximation of the full data. It loses the ability to tell, for example, precisely how much code each contributor has written, but the number of commits is an acceptable substitute.

The data obtained was a total of 521 MB in size, measured as SQL statements. Before deciding on the exact values to present, we explored the data using data mining techniques and simple graphing of the variables over time. We focused primarily on the correlations between different data items and time-line analysis of single and multiple variables. We used the model in Figure 6 to guide our exploration.

6 Experiment: Results

The collected data can be used as a basis for a wide variety of quality-related observations. In the following sections, we will present metric observations that highlight some of the important quality factors in our quality model for each of the three selected projects. We find that the three FOSS projects vary in the amount of contributors involved and the amount of code produced and bugs submitted, but that

Metric family	Metric source	Metric code	Metric description
CCCC	All code	SLOC	Source lines of code
		CLOC	Comment lines
		REJLOC	Lines rejected by CCCC
		CYC	Cyclomatic complexity
		IF	Information flow
		NOM	Number of modules
	Per module	CBO	Coupling between object classes
		DIT	Depth of inheritance tree
		WMC	Weighted methods per class
		FANIN	Fan-in (producer role)
		FANOUT	Fan-out (consumer role)
		IF	Information flow
Static C and static Python	All and per module	SLOC	Source lines of code
		PLOC	Physical lines of code
		BLOC	Blank lines of code
		CLOC	Comment lines
		NOM	Number of modules
		CYC	Cyclomatic complexity
		FANIN	Fan-in (function parameters)
		FANOUT	Fan-out (return statements)
Diff statistics	All code	FC	Files changed
		LOC_ADD	Lines of code added
		LOC_DEL	Lines of code deleted
File types	All files	FTYPE	MIME type of file
		COMP	Compression type, if any
Version control log	Revision log	AUTHOR	Author of the revision
		TIMESTAMP	Timestamp of the revision
		REVID	Native revision identifier

Table 10: Description of calculated metrics.

these are not decisive when determining the quality of the end result. However, some patterns are visible that are typical of FOSS projects.

6.1 Linux

Linux is an operating system kernel initially written in 1991 by Linus Torvalds, at the time a student at the University of Helsinki. The kernel was written by replacing parts of the educational Minix system until none of the original parts remained. Since then, each part of the kernel has been rewritten or extended, and very little remains of the original Linux version 1.0. Linux is used worldwide for a variety of tasks. The project has worked on the 2.6 branch since December 2003. Our data begins in January 2000 for the mailing lists, in October 2002 for the bugs, and in April 2002 for the source code. We will make comparisons with the Linux-historical data where appropriate.

6.1.1 Process quality

The number of commits per author follows a power law, with most commits being contributed by a small number of authors (Figure 7). Estimating the number of commits for core developers at 500 or more, and for co-developers at 50 or more, we find that the core developers constitute approximately half a percent of all committers, and co-developers slightly more than 6% (Table 11). It should be noted that this definition of developer classes is only an approximation of the socially assigned roles within the project.

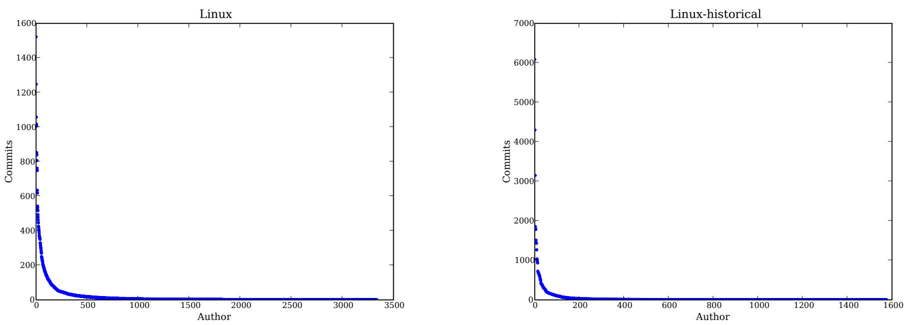


Figure 7: Commits per author for Linux (a) and Linux-historical (b).

Developer class	Size: Linux	Size: Linux-historical	Size: difference	Percentage of committers: Linux	Percentage of committers: Linux-historical	Percentage of committers: difference
Core developers ($\geq$ 500 commits)	18	26	-8	0.54%	1.65%	-1.11%
Co-developers ($\geq$ 50 commits)	213	125	88	6.38%	7.93%	-1.55%
Active users	3106	1426	1680	93.08%	90.42%	2.66%

Table 11: Developer classes in Linux and Linux-historical.

Compared to Linux-historical, the core team has shrunk by more than one percentage unit, and the co-developers set has shrunk by slightly more than one and a half percentage units. The active user set has in turn grown by more than two and a half percentage units. This could indicate that there is a trend toward more distributed development, where a shrinking core is moderating a growing mass of changes. The quality implication is that the use of the Git tool has enabled development to become more distributed, reducing the load on the core and co-developers, but also increasing the need for a hierarchical peer-review process.

We made no attempt to estimate the number of passive users and readers. The former is likely to range in the millions, and the latter in at least the thousands. Therefore, they constitute a significant positive contribution to the organisational quality; there is a very large number of external observers.

We detected a total of 22 236 distinct persons posting to the Linux kernel developers' mailing list, and a total of 646 001 messages. The same kind of distribution applies to mailing list participation and bug submission as to source code commits (Figure 8). These findings are consistent with the studies presented earlier; a majority of all work is done by a small number of contributors, while the majority of contributors contribute only once. There are no special quality implications compared to other FOSS projects other than the scale of the project.

6.1.2 Product quality

The SLOC evolution graph shows that Linux 2.6 has been growing at a linear rate. The same is true for Linux-historical (Figure 9). In light of this data, we observe that there is no evidence for the super-linear growth found by Godfrey and Tu [GoT00, GoT01]. They formulated the polynomial $y = 0.21x^2 + 252x + 90\,055$ to model the uncommented SLOC number y based on the number of days x since the

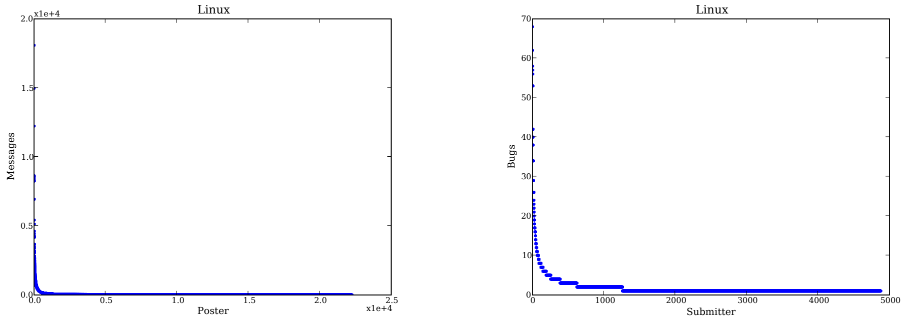


Figure 8: Posts per poster (a) and bugs per submitter (b) for Linux.

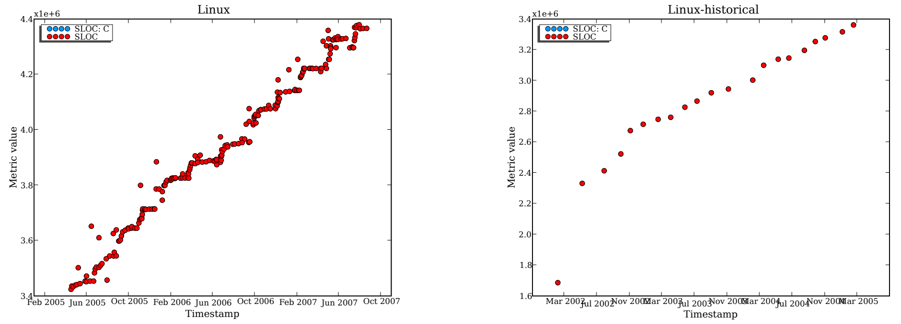


Figure 9: SLOC evolution for Linux (a) and Linux-historical (b).

release of Linux 1.0 on the 13th of March 1994. Given that x is now approximately 4900, y should be approximately 6 366 955, but the current size is almost two million lines short of that (Figure 9(a)). Perhaps the time period measured by Godfrey and Tu was an initial growth phase. As seen today, Linux has been following the same linear growth rate as industrial software [Leh97]. This is also true at the subsystem level; the great majority of source code is in the drivers subsystem, which also exhibits the fastest growth, but not a super-linear one (Figure 10).

Thus, our findings give no reason to believe that the process employed by Linux is of superior quality when sustained super-linear growth is desired. However, the data does show that the Linux project has been able to keep its pace despite an increasingly large and complex code base.

The commit frequency shows that activity has been varying between a few hundred commits to nearly 4000 commits per 30-day interval (Figure 11(a)). It is difficult to detect a trend, but the activity does seem to be cyclic, more or less following the

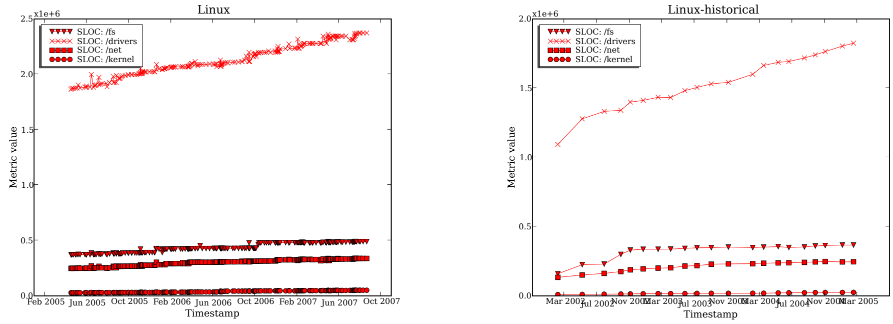


Figure 10: Subsystem growth in Linux (a) and Linux-historical (b).

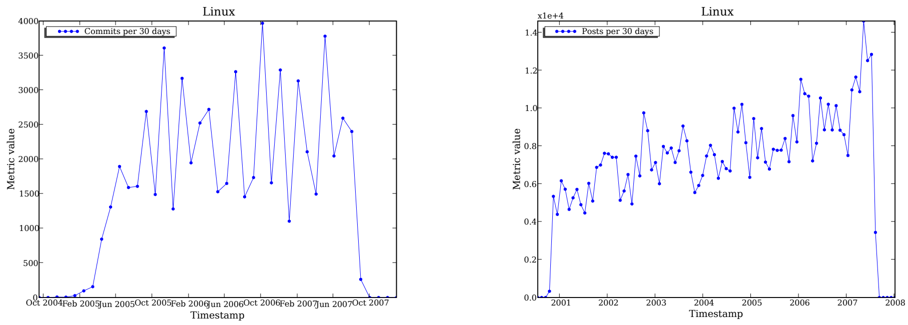


Figure 11: Commit (a) and post (b) frequency for Linux.

development phases in use in the project. The post frequency shows a rising trend (Figure 11(b)). Discussion seems to be increasing, which could be a sign that more design work and coordination is performed as the code base grows. Alternatively, it could be a result of the steps taken to distribute development – as the barrier of participation decreases, the pace of discussion increases.

The bug arrival rate shows no consistent trends (Figure 12(a)). It appears that the bug arrival rate has been declining during the year 2007, and that a similar decline occurred from 2004 to mid-2005. The bug arrival rate per commit rate shows that the Linux project is capable of countering the amount of reported bugs without problems (Figure 12(b)). We note that the Linux project is somewhat divided in the use of the Bugzilla bug tracking system, as the project has traditionally communicated bug reports on the mailing list or directly between users and subsystem maintainers, a practice still in active use. Also, Linux distributions carry a significant percentage of the bugs originating from end-users in their own bug tracking systems, and thus only a part of the bugs exist in the bug tracking system of the Linux project.

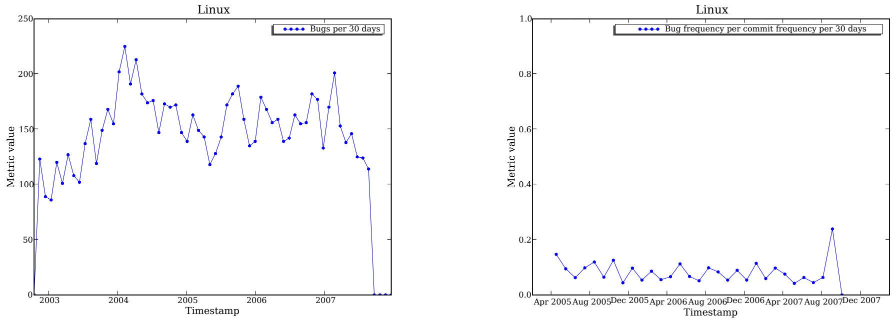


Figure 12: Bug frequency (a) and arrival rate per commit rate (b) for Linux.

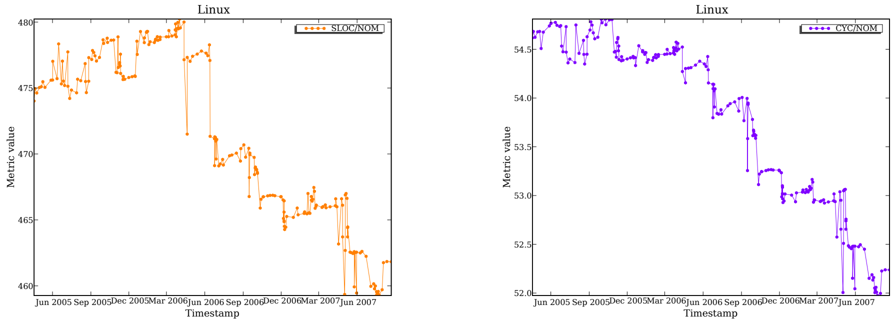


Figure 13: SLOC per NOM (a) and CYC per NOM (a) for Linux.

Linux is becoming more modular, with a decreasing average module size (Figure 13(a)). Also, the cyclomatic complexity is decreasing (Figure 13(b)). This indicates that the Linux project is taking steps to ensure maintainability.

Finally, the fan-in and fan-out per source lines of code is slowly increasing (Figure 14). This indicates that there is either an increased number of small functions, or the functions accept more parameters. Either of these is an indication of work toward more maintainable code, since an increased number of small functions means each function performs a more specific, well-defined task, and more parameters indicates more general functions that can be reused in a larger number of cases. Detailed code inspection would be required to establish the exact details, however.

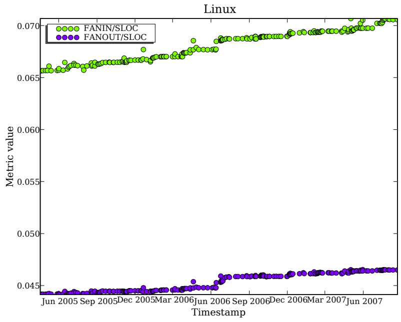


Figure 14: Fan-in and fan-out per SLOC for Linux.

6.2 Blender

Blender is a 3D content creation suite. It includes modelling, animation, shading, rendering, a physics and particle system, imaging and compositing facilities, and real-time 3D and game creation functions. Blender was initially an in-house tool developed for an animation studio. The tool was spun off in 1998 to a separate company which was later shut down because of economical difficulties. In 2002, the newly formed Blender Foundation bought the source code using donated funds from the FOSS community. Since then, Blender has been developed as a FOSS project.

6.2.1 Process quality

Figure 15 shows the commits per author, posts per poster and bugs per submitter for the Blender project. The latter two display the usual power law distribution common in FOSS projects, but the first has some unusual properties. The number of code-submitting actors in the project is smaller than what might be expected from the code size and complexity of the project. Less than 70 persons have committed code into the repository. Using the same definitions as for Linux, we see that the number of core developers is only two, and there is a surprisingly large amount of co-developers (Table 12).

We believe that this is consistent with the very modular nature of the program. Blender acts as a framework for all 3D content creation tasks – tasks which are related within their field, but which result in very different implementation details and requirements of mathematical knowledge. The large co-developer class may indicate that developers of specific sub-features work outside the Blender repository to implement the desired functionality, committing them in relatively few steps.

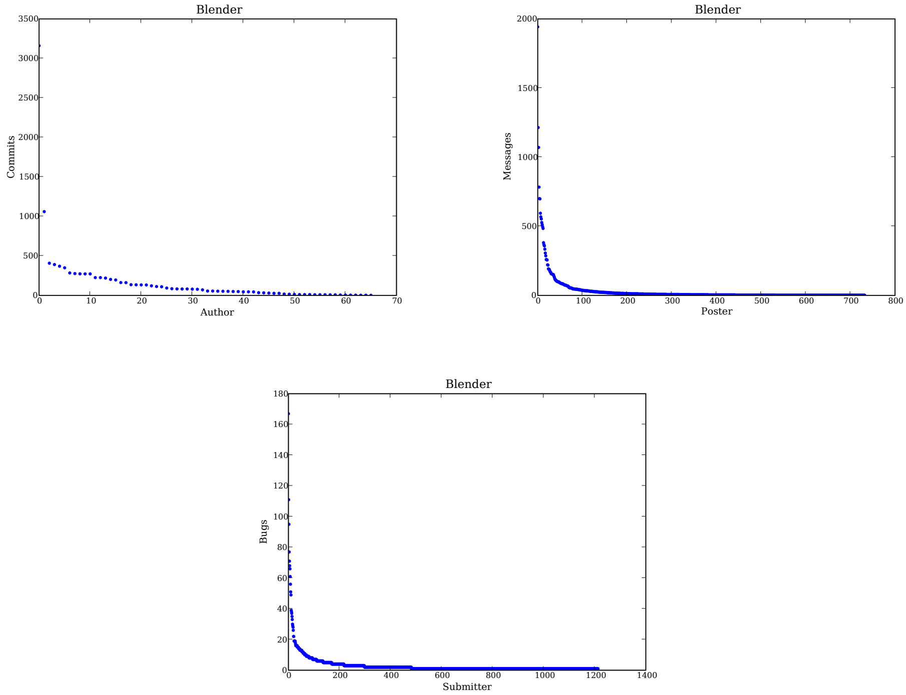


Figure 15: Commits per author (a), posts per poster (b) and bugs per submitter (c) for Blender.

Developer class	**Size**	**Percentage of committers**
Core developers ($\geq$ 500 commits)	2	3.03%
Co-developers ($\geq$ 50 commits)	36	54.55%
Active users	28	42.42%

Table 12: Developer classes in Blender.

We detected a total of 733 distinct persons posting to the two Blender development lists, one for the main program development and the other for Python-related development. A total of 23 891 messages were detected. We observe that Blender has a highly specialised and small audience participating in the visible activities. We suspect that the activity is higher in the Blender web forums, but also that there is a very high number of passive users compared to the other actor classes. Blender is directed toward artistic users who may not have an interest in participating in traditional FOSS activities. This is also visible when comparing the amount of commits, posts, and bugs – the last is perhaps the most anonymous and quick form of feedback, and the amount of bug submitters is roughly 30% larger than the amount of mailing list posters.

We conclude that communication in Blender is quite streamlined, and we suspect that much of it occurs outside the observed data sources. The organisation is small enough to work without complicated processes, and the challenge in this project lies in the knowledge of 3D graphics and 3D content production required to write this kind of program.

6.2.2 Product quality

The SLOC evolution of Blender shows a steady increase in the amount of Python code (Figure 16(a)). It also shows a carefully super-linear growth until mid-2006, after which there is a large, sudden increase in code size. After that, SLOC growth continues, but at a decreased pace.

Examining the subsystem-level SLOC evolution reveals that the sudden increase in code size is due to two external libraries being imported into the Blender VCS: ffmpeg, a collection of libraries to handle digital audio and video, and Verse, a library for sharing 3D data over a network. Figure 16(b) shows the impact of these, and also reveals that the SLOC evolution in the main Blender code continues its super-linear growth. The recent decrease in total SLOC count likely results from external libraries being shrunk or removed. We thus have reason to believe that the project is highly efficient and can produce code at a rapid pace. Based on this data, the reason appears to be that Blender is a small enough project to avoid unnecessary communication overhead, that the developers are well synchronised, and that there is a high degree of knowledge among the project members.

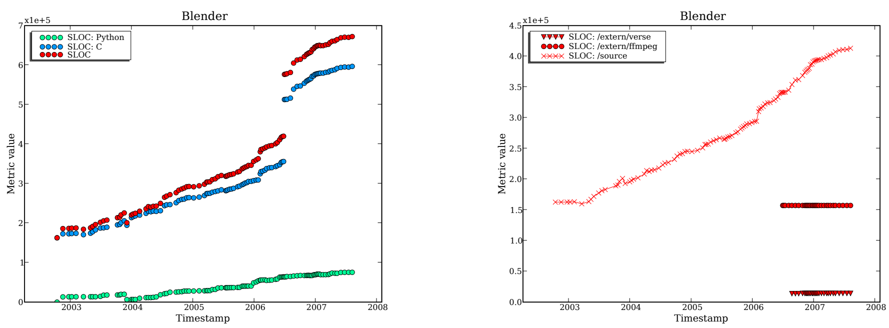


Figure 16: SLOC evolution (a) and SLOC evolution for selected subsystems (b) in Blender.

The commit frequency shows that activity has been varying between less than 100 to approximately 450 commits per 30-day interval (Figure 17(a)). The activity has slowly increased, but during 2007, it has decreased. The post frequency shows a more or less constant activity (Figure 17(b)). We believe that more activity occurs in the Blender web fora.

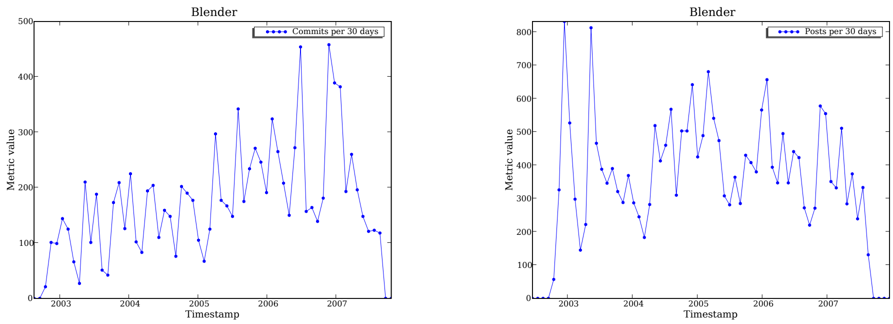


Figure 17: Commit (a) and post frequency (b) for Blender.

The bug arrival rate is increasing, and its variation is also increasing (Figure 18(a)). We suspect that the Blender user community is growing, and that each new release attracts more sporadic bug submitters. The bug frequency per commit frequency shows that Blender is able to work faster than the bug submissions arrive, giving a careful indicator that the project is capable of addressing the issues brought up by users (Figure 18(b)).

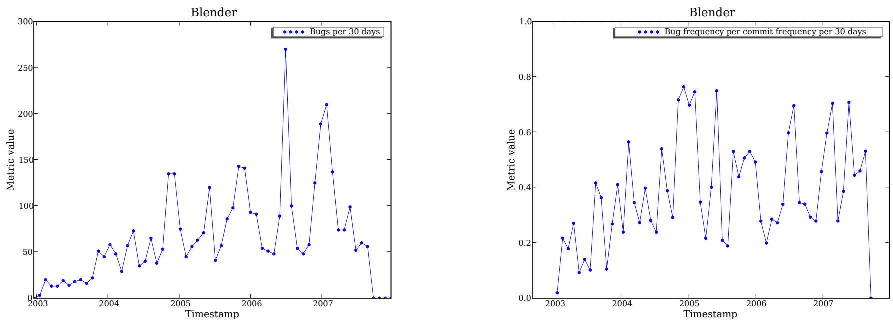


Figure 18: Bug arrival rate (a) and arrival rate per commit rate (b) for Blender.

Blender has a tendency to become less modular, but regular decreases in the SLOC per NOM metric indicates that the project may refactor the code to reduce this tendency (Figure 19(a)). The same is seen in the cyclomatic complexity distribution over modules (Figure 19(b)).

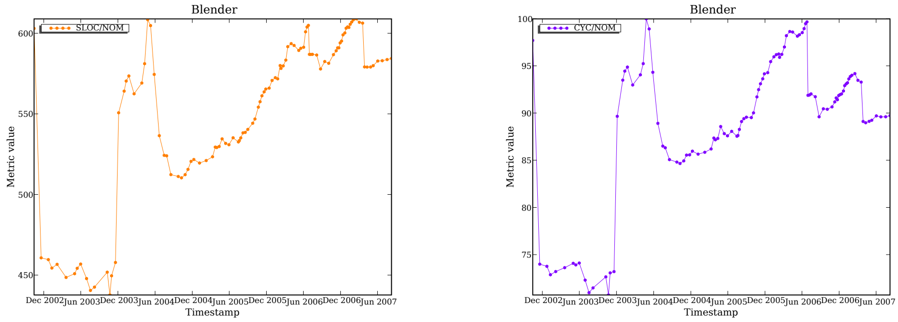


Figure 19: SLOC per NOM (a) and CYC per NOM (b) for Blender.

Finally, the fan-in per SLOC is slowly increasing while the fan-out per SLOC has begun decreasing slightly in the second quarter of 2006 (Figure 20). This could indicate that functions are becoming more generic, accepting a greater number of parameters to control their behaviour, while the project is attempting to lower the complexity of the program by reducing the number of exit points from each function. However, since the decrease in fan-out appears to level out, this might be a pure coincidence.

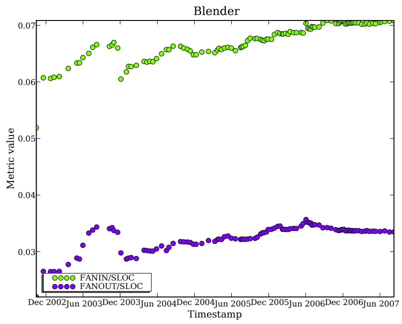


Figure 20: Fan-in and fan-out per SLOC for Blender.

6.2.3 Object-oriented features

Since Blender is written largely in C++, this project was the most suitable to apply the object-oriented metrics on. The growth of information flow indicates that the coupling between classes has increased (Figure 21(a)). We can see that the DIT and NOC metrics react to the same changes: as the depth of the inheritance tree increases, the number of classes increases by the same relative amount (Figure

21(b)). These figures indicate that Blender is becoming more modular and that there is greater potential for code reuse. The flexibility of the code increases, but as a result, understandability could decrease.

The increasing WMC metric shows that the functionality of all classes has increased (Figure 21(c)). We also see correlation between CBO and fan-in – they measure essentially the same thing. Coupling between objects seems to have increased moderately over time, but the growth curve has stopped increasing in 2007, indicating that the class hierarchy has stabilised.

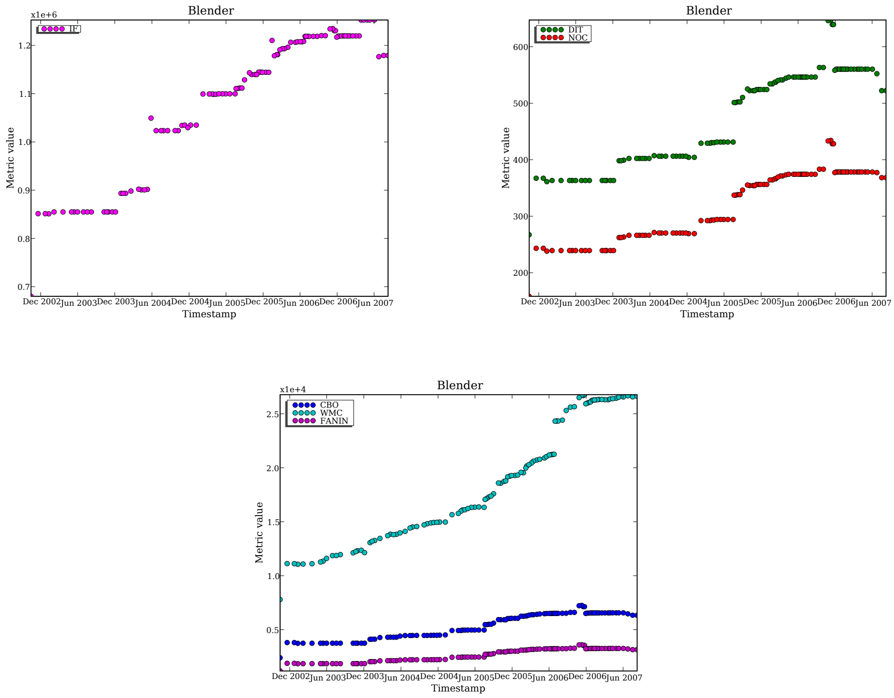


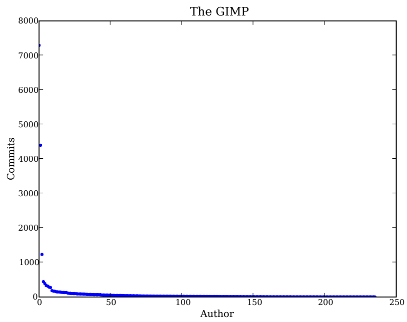
Figure 21: Information flow (a), DIT and NOC (b), and CBO, WMC, and fan-in (c) for Blender.

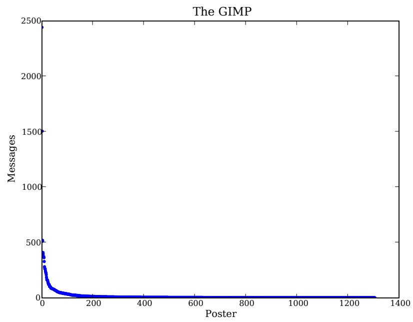
6.3 The GIMP

The GIMP, also known as GIMP, is a graphics program for tasks such as photo retouching, image composition and image authoring. It can be used as a simple paint program, an on-line batch processing system, a mass production image renderer, or an image format converter. Other uses are possible through scripting and plug-in mechanisms. GIMP has been in development since 1996.

6.3.1 Process quality

The commits per author, posts per poster and bugs per submitter for the GIMP project all display the usual power law distribution common in FOSS projects, the first having an unusually steep slope for the first few authors (Figure 22). The number of posters is more than five times as many as the code contributors, while the number of bug submitters is another two and a half times more numerous. Using the same definitions as for Linux, we see that the number of core developers is three, the number of co-developers is 47 and the number of active users is 186 (Table 13).





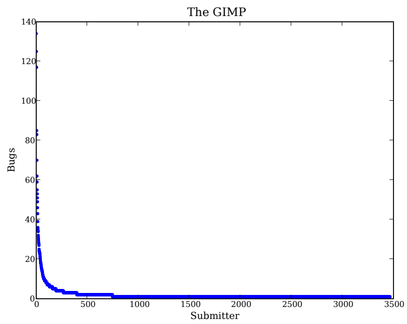


Figure 22: Commits per author (a), posts per poster (b) and bugs per submitter (c) for GIMP.

Developer class	Size	Percentage of committers
Core developers ($\geq$ 500 commits)	3	1.27%
Co-developers ($\geq$ 50 commits)	47	19.92%
Active users	186	78.81%

Table 13: Developer classes in GIMP.

We detected a total of 1306 distinct posters on the GIMP developer mailing list, contributing to a total of 20 320 messages. This is not surprising given the age of the project.

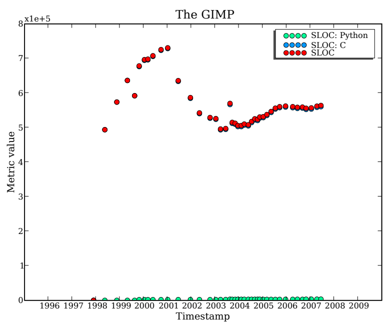
We conclude that GIMP is quite a typical FOSS project, consistent with the hypotheses put forward by Mockus et al. [Moc02]. In particular, the fourth hypothesis has strong support: GIMP has successfully gained contributors beyond the core developers, and has thus been able to sustain itself over a period of nearly ten years.

6.3.2 Product quality

The SLOC evolution of GIMP shows that code size increased sub-linearly until 2001, after which it returned to roughly its earlier size (Figure 23(a)). After that, growth has been more controlled, but approximately linear. One reason for this may be that much of the user interface code written for the program has been broken out into a separate GUI library called GTK+, and new functionality has gradually moved from the GIMP project to GTK+.

Examining the subsystem-level SLOC evolution shows that a significant portion of the code consists of the main application and the plug-ins. Another major part of the code is the `libgimp` library (Figure 23(b)).

The commit frequency shows that activity has been varying between less than 100 to almost 500 commits per 30-day interval (Figure 24(a)). The activity appears to have two phases, each being a rapid increase followed by a longer decrease. The post frequency shows a more or less constant activity (Figure 24(b)). The anomaly



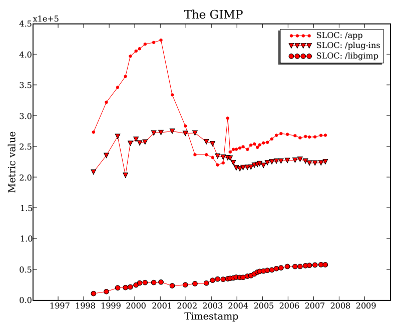


Figure 23: SLOC evolution (a) and SLOC evolution for selected subsystems (b) in GIMP.

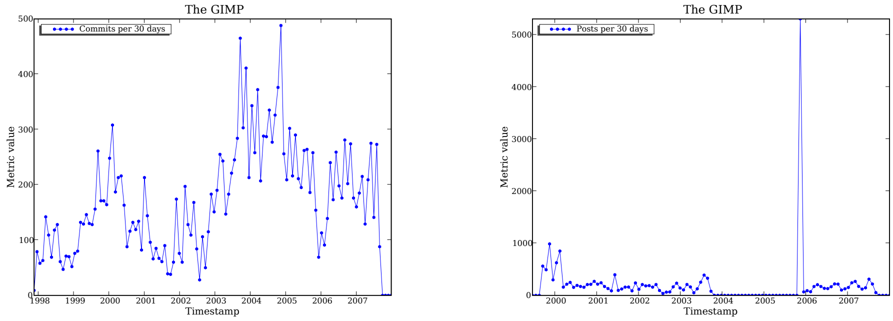


Figure 24: Commit (a) and post frequency (b) for GIMP.

at the end of 2005 suggests that the time stamps of a large number of messages in 2004 to 2005 have been displaced to that period.

The bug arrival rate has begun decreasing after a long period of increase (Figure 25(a)). We can apply the idea of stabilisation time to this observation: the program has reached a level of maturity, and as a result, bug reports are decreasing. However, the arrival rate has not yet reached a steady level. Also, work is ongoing to change the program to be more similar to other, more modern paint programs. This could lead to an increased bug arrival rate as new code is deployed by users.

The bug frequency per commit frequency shows that GIMP is mostly able to work faster than the bug submissions arrive (Figure 25(b)). However, even recently, the rate of bug submissions has momentarily been higher than the commit frequency, suggesting that there is sometimes a surge in bug arrival rates that the project is not able to counter. However, this is compensated later.

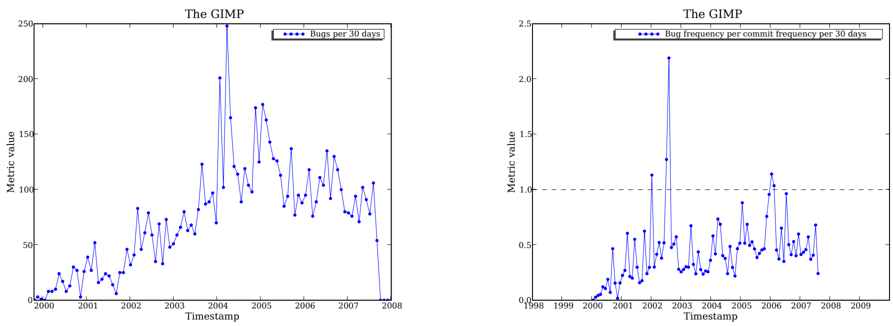


Figure 25: Bug frequency (a) and arrival rate per commit rate (b) for GIMP. The dashed line indicates the point where the bug arrival rate exceeds the commit rate.

GIMP has become more modular (Figure 26(a)). However, it may be reaching its practical lower limit for SLOC per number of modules, since the decrease in this number has slowed down. The cyclomatic complexity per module reflects exactly the same pattern, suggesting that the code itself has not become less complex, but has been split up into more manageable parts (Figure 26(b)).

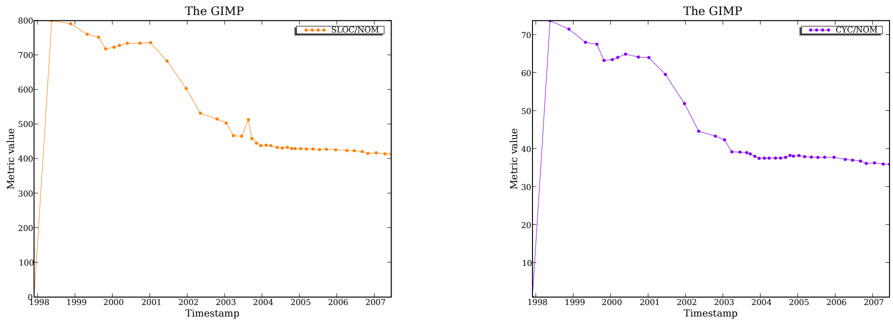


Figure 26: SLOC per NOM (a) and CYC per NOM (b) for GIMP.

Finally, the fan-in per SLOC is slowly increasing while the fan-out per SLOC is more or less constant (Figure 27). This indicates that the project has not observed a need to simplify the interface of functions.

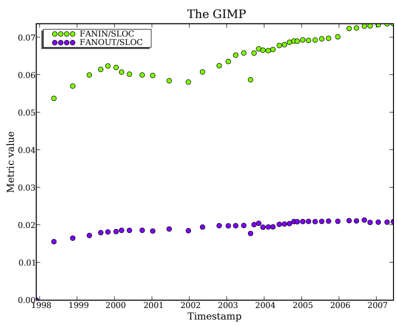


Figure 27: Fan-in and fan-out per SLOC for GIMP.

6.4 Summary

Our experiment shows that the quality model described in Section 4 can be applied to real FOSS projects. Both general software quality factors and FOSS-specific factors contribute to the overall quality assessment in our experiment (Table 14). The metric trends are either increasing or decreasing, or they follow a repeated cycle of increase and decrease over some time interval. In Linux, increased distribution of development poses significant challenges, but our data shows that the project is handling this well. Increased discussion could be a sign of growing needs of coordination. However, the team roles have adjusted to include more hierarchical moderation. In Blender, there is a quality risk with regard to the program architecture; complexity is increasing and modularity is decreasing. The source code growth is super-linear, which may indicate that the project is undergoing heavy development compared to the other two projects. The GIMP appears to have moved to the maintenance phase, since the bug arrival rate has been decreasing for three years after a four-year period of increase. The project may have to turn resources toward handling bug reports, especially if new features are to be introduced in the future.

	Linux	**Linux-historical**	**Blender**	**The GIMP**
Core developers ($\geq$ 500 commits)	18 (0.54%)	26 (1.65%)	2 (3.03%)	3 (1.27%)
Co-developers ($\geq$ 50 commits)	213 (6.38%)	125 (7.93%)	36 (54.55%)	47 (19.92%)
Active users	1306 (93.08%)	1426 (90.42%)	28 (42.42.%)	186 (78.81%)
Posters	22 236	–	733	1306
Messages	646 001	–	23 891	20 320
Growth	linear	linear	super-linear	sub-linear / linear
Bug submitters	4877	–	1214	3468
Bugs	8954	–	4048	7232
Total SLOC	4 366 435 (21.8.2007)	3 361 971 (15.2.2005)	596 815 (6.8.2007)	560 408 (12.6.2007)
Commit frequency trend	cyclic	cyclic	slow increase	cyclic
Post frequency trend	increasing	–	constant	constant
Bug arrival rate trend	cyclic	–	increasing	decreasing (after increase)
Bug frequency per commit frequency trend	constant low ($\leq$ 0.25)	–	increasing moderate ($\leq$ 0.8)	constant moderate ($\leq$ 1.6, most samples $\leq$ 0.8)
Modularity	increasing	increasing (some variation)	decreasing	increasing
Complexity	decreasing	decreasing	increasing	decreasing

Table 14: Summary and interpretation of experiment results.

7 Conclusions

We have presented a theoretical background for software quality, metrics, and their application in a FOSS environment. We have shown that a vast amount of information is available from FOSS projects in three information spaces and we have constructed a quality model suitable for use in a FOSS context. Finally, we have applied a subset of this model to three FOSS projects and highlighted both theoretical and practical concerns in implementing automatic metric collection and analysis.

FOSS projects are faced with challenges seldom seen in traditional software development projects. The scale at which many FOSS projects are operating belongs to the high end of software development in terms of participant numbers, code size, and problem complexity. The cultural and practical barriers that have to be overcome are significant, as are the requirements on leadership and administration. Therefore, we believe that FOSS projects can benefit from automatically collected data that allows a higher-level view of progress than what is currently available.

We identify three core challenges that the FOSS community must overcome to improve quality processes. First, the tools used in FOSS development must be fitted with features that allow data to be extracted in sensible formats. Currently, one of the most popular bug tracking systems, Bugzilla, lacks features to remotely export the entire bug event history in a machine-readable format. Similarly, the overhead required to download the entire event history from the Subversion version control system is significant, as each operation requires network communication with the main server. The GCC compiler would be a natural location to place source code metric calculation facilities, since it already implements a semantic parser for many programming languages. Also, it benefits from information in the software build system. Writing a completely new tool to extract all this information would be a significant task.

Second, as FOSS has a low barrier to adoption of new technologies – we can safely say that FOSS creates most of its own technology – metric tools for these are needed. Logic, dynamic, and interpreted languages such as Python and Ruby have enabled accelerated development through higher expressive power, reusable code libraries, and in-language support for streamlined development and test cycles, but there are few tools and methods to assess the complexity of programs written in these languages. There is also little knowledge of the architecture and design patterns that these languages encourage. In the same way as object-oriented design is different from procedural design, these languages encourage a different cognitive approach. The quality of design cannot be assessed without further knowledge of the patterns that these languages enable.

Finally, the FOSS culture has traditionally eschewed rigid processes and management-by-numbers, relying more on established experience and experimentation. In order to introduce quality concepts and metrics-based evaluation, FOSS practitioners must learn about these and see their benefit in assisting their work. At best, FOSS projects work as finely tuned engines, and any change in work flow must ensure that there are no bottlenecks or single points of failure.

The most important factor for enabling quality in FOSS lies without doubt in the transparent, publicly visible development method and the code of conduct that encourages sharing of knowledge in all its forms. Without these traits, FOSS projects would cease to experiment, their communities of users and developers would collapse, and they would become obsolete. An understanding of this value system must be present in any attempts to enhance the quality of Free and Open Source Software projects.

References

AsB02 Asklund, U. and Bendix, L., A Study of Configuration Management in Open Source Software Projects. *IEEE Software*, 149, pages 40–46.

Bou06 Bouktif, S., Antoniol, G. and Merlo, E., A feedback based quality assessment to support open source software evolution: the GRASS case study. *Proceedings of the 22nd IEEE international conference on software maintenance*, Washington, DC, USA, 2006, IEEE Computer Society, pages 155–165.

Boe76 Boehm, B. W., Brown, J. R. and Lipow, M., Quantitative evaluation of software quality. *Proceedings of the 2nd international conference on software engineering*, Los Alamitos, CA, USA, 1976, IEEE Computer Society Press, pages 592–605.

Bar05 Barcellini, F., Détienne, F., Burkhardt, J.-M. and Sack, W., Thematic coherence and quotation practices in OSS design-oriented online discussions. *Proceedings of the 2005 international ACM SIGGROUP con-*

ference on supporting group work, New York, NY, USA, 2005, ACM Press, pages 177–186.

Cro04 Crowston, K., Annabi, H., Howison, J. and Masango, C., Effective work practices for software engineering: free/libre open source software development. *Proceedings of the 2004 ACM workshop on interdisciplinary software engineering research*, New York, NY, USA, 2004, ACM Press, pages 18–26.

CrH06 Crowston, K. and Howison, J., Assessing the health of open source communities. *IEEE Computer*, 39,5(2006), pages 89–91.

ChK94 Chidamber, S. R. and Kemerer, C. F., A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20,6(1994), pages 476–493.

Coo82 Cook, M. L., Software metrics: an introduction and annotated bibliography. *SIGSOFT Software Engineering Notes*, 7,2(1982), pages 41–60.

Chr89 Christodoulakis, D., Tsalidis, C., van Gogh, C. and Stinesen, V., Towards an automated tool for software certification. *IEEE international workshop on tools for artificial intelligence*, Fairfax, VA, USA, 1989, IEEE, pages 670–676.

Cha06 Chan, V. K. Y., Wong, W. E. and Xie, T. F., Application of a statistical methodology to simplify software quality metric models constructed using incomplete data samples. *Proceedings of the 6th international conference on quality software*, Washington, DC, USA, 2006, IEEE Computer Society, pages 15–21.

DiS06 Dick, S. and Sadia, A., Fuzzy clustering of open-source software quality data: a case study of Mozilla. *International Joint Conference on Neural Networks*, Vancouver, BC, Canada, 2006, pages 4089–4096.

DiB04 Dinh-Trong, T. and Bieman, J. M., Open source software development: a case study of FreeBSD. *Proceedings of the 10th international symposium on software metrics*, Washington, DC, USA, 2004, IEEE Computer Society, pages 96–105.

DiB05 Dinh-Trong, T. and Bieman, J. M., The FreeBSD project: a replication case study of open source development. *IEEE Transactions on Software*

Engineering, 31,6(2005), pages 481–494. Senior Member-James M. Bieman.

FeN00 Fenton, N. E. and Neil, M., Software metrics: roadmap. *Proceedings of the conference on the future of software engineering*, New York, NY, USA, 2000, ACM Press, pages 357–370.

Fre07 Free Software Foundation Inc., The Free Software Definition, Www, 14 June 2007. URL `http://www.gnu.org/philosophy/free-sw.html`.

Fer04 Ferenc, R., Siket, I. and Gyimothy, T., Extracting facts from open source software. *Proceedings of the 20th IEEE international conference on software maintenance*, Washington, DC, USA, 2004, IEEE Computer Society, pages 60–69.

Gaf81 Gaffney, J. E. J., Metrics in software quality assurance. *Proceedings of the ACM '81 conference*, New York, NY, USA, 1981, ACM Press, pages 126–130.

Gre03 Greiner, S., Boskovič, B., Brest, J. and Žumer, V., Security issues in information systems based on open-source technologies. *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, 2, pages 12–15.

Gyi05 Gyimothy, T., Ferenc, R. and Siket, I., Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions of Software Engineering*, 31,10(2005), pages 897–910.

GoT00 Godfrey, M. and Tu, Q., Evolution in open source software: A case study. *Proceedings of the international conference on software maintenance*, Washington, DC, USA, 2000, IEEE Computer Society, page 131.

GoT01 Godfrey, M. and Tu, Q., Growth, evolution, and structural change in open source software. *Proceedings of the 4th international workshop on principles of software evolution*, New York, NY, USA, 2001, ACM Press, pages 103–106.

Hal75 Halstead, M. H., Toward a theoretical basis for estimating programming effort. *Proceedings of the 1975 ACM annual conference*, New York, NY, USA, 1975, ACM Press, pages 222–224.

HoK97 Houdek, F. and Kempter, H., Quality patterns – an approach to packaging software engineering experience. *Proceedings of the 1997 symposium on software reusability*, New York, NY, USA, 1997, ACM Press, pages 81–88.

HaO01 Hars, A. and Ou, S., Working for free? – motivations of participating in open source projects. *Proceedings of the 34th annual Hawaii international conference on system sciences*, volume 7, Washington, DC, USA, 2001, IEEE Computer Society, page 7014.

Hun03 Huntley, C. L., Organizational learning in open-source software projects: An analysis of debugging data. *IEEE Transactions on Engineering Management*, 50,4(2003), pages 485–493.

IzB06 Izurieta, C. and Bieman, J., The evolution of FreeBSD and Linux. *Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering*, New York, NY, USA, 2006, ACM Press, pages 204–211.

IEE04 IEEE Computer Society, New York, NY, USA, *IEEE Standard for a Software Quality Metrics Methodology (IEEE Std 1061)*, R2004 edition, 24 June 2004.

Kaf85 Kafura, D., A survey of software metrics. *Proceedings of the 1985 ACM annual conference on the range of computing: mid-80's perspective*, New York, NY, USA, 1985, ACM Press, pages 502–506.

KoS00 Koch, S. and Schneider, G., Results from Software Engineering Research into Open Source Development Projects Using Public Data. Discussion paper for Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft, Wirtschaftsuniversität Wien, 2000.

Kho95 Khoshgoftaar, T. M., Szabo, R. M. and Guasti, P. J., Exploring the behaviour of neural network software quality models. *Software Engineering Journal*, 10, pages 89–96.

Li05 Li, P. L., Herbsleb, J. and Shaw, M., Finding predictors of field defects for open source software systems in commonly available data sources: a case study of OpenBSD. *Proceedings of the 11th IEEE international software metrics symposium*, Washington, DC, USA, 2005, IEEE Computer Society, page 32.

Lin04 Lindman, J., Effects of open source software on the business patterns of software industry. Master's thesis, Helsinki School of Economics, 2004.

Leh97 Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E. and Turski, W. M., Metrics and laws of software evolution - the nineties view. *Proceedings of the 4th international symposium on software metrics*, Washington, DC, USA, 1997, IEEE Computer Society, page 20.

Lan05 Langelier, G., Sahraoui, H. and Poulin, P., Visualization-based analysis of quality for large-scale software systems. *Proceedings of the 20th IEEE/ACM international conference on automated software engineering*, New York, NY, USA, 2005, ACM Press, pages 214–223.

Liu07 Liu, Y., Yao, J.-F., Williams, G. and Adkins, G., Studying software metrics based on real-world software systems. *Journal of Computing in Small Colleges*, 22,5(2007), pages 55–61.

MäM06 Mäki-Asiala, P. and Matinlassi, M., Quality assurance of open source components: Integrator point of view. *Proceedings of the 30th annual international computer software and applications conference*, Washington, DC, USA, 2006, IEEE Computer Society, pages 189–194.

Mas05 Massey, B., Longitudinal analysis of long-timescale open source repository data. *Proceedings of the 2005 workshop on predictor models in software engineering*, New York, NY, USA, 2005, ACM Press, pages 1–5.

Moc02 Mockus, A., Fielding, R. T. and Herbsleb, J. D., Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11,3(2002), pages 309–346.

Mic05a Michlmayr, M., Hunt, F. and Probert, D., Quality practices and problems in free software projects. *Proceedings of the First International Conference on Open Source Systems*, Scotto, M. and Succi, G., editors, Genova, Italy, 2005, pages 24–28.

Mic05b Michlmayr, M., Quality improvement in volunteer free software projects: Exploring the impact of release management. *Proceedings of the First International Conference on Open Source Systems*, Scotto, M. and Succi, G., editors, Genova, Italy, 2005, pages 309–310.

Mic05c Michlmayr, M., Software process maturity and the success of free software projects. *Software Engineering: Evolution and Emerging Technologies*, Zielinski, K. and Szmuc, T., editors, Krakow, Poland, 2005, IOS Press, pages 3–14.

Mic07 Michlmayr, M., *Quality Improvement in Volunteer Free and Open Source Software Projects: Exploring the Impact of Release Management*. Ph.D. thesis, University of Cambridge, March 2007.

Nag04 Nagappan, N., Toward a software testing and reliability early warning metric suite. *Proceedings of the 26th international conference on software engineering*, Washington, DC, USA, 2004, IEEE Computer Society, pages 60–62.

Nag05 Nagappan, N., Williams, L., Vouk, M. and Osborne, J., Early estimation of software quality using in-process testing metrics: a controlled case study. *Proceedings of the third workshop on software quality*, New York, NY, USA, 2005, ACM Press, pages 1–7.

Nak02 Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K. and Ye, Y., Evolution patterns of open-source software systems and communities. *Proceedings of the international workshop on principles of software evolution*, New York, NY, USA, 2002, ACM Press, pages 76–85.

Ola92 Olagunju, A. O., Concepts of operational software quality metrics. *Proceedings of the 1992 ACM annual conference on communications*, New York, NY, USA, 1992, ACM Press, pages 301–308.

Ope07 Open Source Initiative, The Open Source Definition, Www, 14 June 2007. URL `http://www.opensource.org/docs/osd`.

Oga96 Ogasawara, H., Yamada, A. and Kojo, M., Experiences of software quality management using metrics through the life-cycle. *Proceedings of the 18th international conference on software engineering*, Washington, DC, USA, 1996, IEEE Computer Society, pages 179–188.

PhA05 Phadke, A. A. and Allen, E. B., Predicting risky modules in open-source software for high-performance computing. *Proceedings of the second international workshop on software engineering for high performance computing system applications*, New York, NY, USA, 2005, ACM Press, pages 60–64.

Pol04 Polančič, G., Horvat, R. V. and Rozman, T., Comparative assessment of open source software using easy accessible data. *Proceedings of the 26th international conference on information technology interfaces*, volume 1, Slovenia, 2004, Faculty of Electrical Engineering and Computer Science, Maribor University, pages 673–678.

Rob06 Robles, G., Gonzalez-Barahona, J. M., Michlmayr, M. and Amor, J. J., Mining large software compilations over time: another perspective of software evolution. *Proceedings of the 2006 international workshop on mining software repositories*, New York, NY, USA, 2006, ACM Press, pages 3–9.

Roy87 Royce, W. W., Managing the development of large software systems: concepts and techniques. *Proceedings of the 9th international conference on software engineering*, Los Alamitos, CA, USA, 1987, IEEE Computer Society Press, pages 328–338. Reprinted from Proceedings, IEEE Wescon, August 1970, pages 1-9.

Roy90 Royce, W., Pragmatic quality metrics for evolutionary software development models. *Proceedings of the conference on TRI-ADA '90*, New York, NY, USA, 1990, ACM Press, pages 551–565.

Sch02 Schneidewind, N. F., Body of knowledge for software quality measurement. *IEEE Computer*, 35,2(2002), pages 77–83.

ShJ06 Sharma, V. S. and Jalote, P., Stabilization time - a quality metric for software products. *Proceedings of the 17th international symposium on software reliability engineering*, Washington, DC, USA, 2006, IEEE Computer Society, pages 45–51.

Sto90 Stockman, S. G., Todd, A. R. and Robinson, G. A., A Framework for Software Quality Measurement. *IEEE Journal on selected areas in communications*, 8,2(1990), pages 224–233.

TaY05 Tamura, Y. and Yamada, S., Comparison of software reliability assessment methods for open source software. *Proceedings of the 11th international conference on parallel and distributed systems – workshops*, Washington, DC, USA, 2005, IEEE Computer Society, pages 488–492.

Vis97 Visaggio, G., Structural information as a quality metric in software systems organization. *Proceedings of the International Conference on Software Maintenance*. IEEE, 1997, pages 92–99.

WeB06 van Wendel de Joode, R. and de Bruijne, M., The organization of open source communities: Towards a framework to analyze the relationship between openness and reliability. *Proceedings of the 39th annual hawaii international conference on system sciences*, Washington, DC, USA, 2006, IEEE Computer Society, page 118.2.

WaH88 Wake, S. A. and Henry, S. M., A model based on software quality factors which predicts maintainability. Technical Report, Department of Computer Science, Virginia Tech, Blacksburg, VA, USA, 1988.

ZhD05 Zhou, Y. and Davis, J., Open source software reliability model: an empirical approach. *Proceedings of the fifth workshop on open source software engineering*, New York, NY, USA, 2005, ACM Press, pages 1–6.