Advances in Software Engineering

# New Generation of Software Metrics

Guest Editors: Giulio Concas, Giovanni Cantone, Ewan Tempero, and Hongyu Zhang

# New Generation of Software Metrics

Advances in Software Engineering

# New Generation of Software Metrics

Guest Editors: Giulio Concas, Giovanni Cantone, Ewan Tempero, and Hongyu Zhang

# Editorial Board

# Contents

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 913892, 2 pages
doi:10.1155/2010/913892

*Editorial*

# New Generation of Software Metrics

**Giulio Concas,[1] Giovanni Cantone,[2] Ewan Tempero,[3] and Hongyu Zhang[4]**

[1] *University of Cagliari, 09124 Cagliari, Italy*
[2] *University of Rome "Tor Vergata", 00133 Rome, Italy*
[3] *The University of Auckland, 1142 Auckland, New Zealand*
[4] *Tsinghua University, Beijing 100084, China*

Correspondence should be addressed to Giulio Concas, siase@diee.unica.it

Received 30 December 2010; Accepted 30 December 2010



Software measurement has always been an issue in software engineering. On one hand, engineering is about designing and building things and modeling, characterizing, monitoring, evaluating, defining, predicting, "prescripting", controlling, and changing processes and their artifacts; additionally, measurement is essential to verify that the built artifacts comply with their requirements, to validate the product built versus its requirements and design and to keep the production process in control; the old management adage "*you can't manage what you don't measure*" is valid also in software development. On the other hand, because software is nonmaterial, it is a very difficult beast to measure. The compliance with requirements, which can vary greatly from system to system even in the same domain, is something difficult to "measure", and it is almost impossible to devise metrics able to support such a measure that is repeatable across different systems. The development effort is perhaps easier to estimate and define across different software technologies and domains—after all it is always a matter of money—but it is still very difficult to measure in the ever changing languages, environments, and technologies.

The difficulty of measuring software is even more critical in present times due to the increasing size and criticality of software within all technological systems, and the frequent changes in architectural paradigms and process models. New development methods, such as agile and lean approaches, the large diffusion of Open Source Software (OSS) products, service oriented architectures, cloud computing and software-as-a-service business models, offshore development, ubiquitous computing, and the increasing integration of software and systems engineering, are examples of how software engineering is coping with the growing complexity of software and its applications. In this scenario, we need even more to be able to measure software products, processes, and competencies.

To this purpose, radically new approaches, methods, and tools are being proposed in the software metrics field. Whereas it is too early to talk of breakthrough innovations and of definitive or also acceptable solutions for the above-cited issues, progress is being made in some interesting directions. Among these, we may mention software product and process metrics derived from complex network theory and in general from theoretical physics studies; social network analysis applied to software products and to developers' relationships; measurement of size, effort, and quality of software developed in innovative ways, such as using agile and lean methodologies and open source projects; new metrics for service-oriented architectures and software-as-a service.

This special issue presents new and relevant research work along these directions. We received 19 submissions and accepted six of them after a thorough peer-review process. These papers cover a broad range of topics in the field of innovative metrics for software, including metrics for OSS maintenance effort and usability, social network analysis applied to software, metrics for model-driven design (MDD), metrics for refactoring, and metrics to assess developers' activity.

In the first paper, titled "*The economics of community open source software projects: an empirical analysis of maintenance effort*," the authors, E. Capra et al. aim to empirically

compare the growth entropy when maintaining OSS and traditional proprietary software applications. They utilize a sample of 4,289 open source application versions. Analyses are based on data obtained by applying a traditional effort estimation model. The open source projects are compared with data obtained from previous empirical software engineering researches. Findings indicate that open source applications show a slower growth of maintenance effort over time and, therefore, are less subject to the entropy effect.

The second paper, "*Improvement of open source software usability: An empirical evaluation from developers' perspective*" by A. Raza et al. presents an empirical investigation to study the impact of five key factors on OSS usability from the developers' point of view. This paper analyzes a data set of 106 OSS developments from 18 OSS projects of different size. The results from this study provide empirical evidence by showing that all the studied key factors significantly improve OSS usability.

In the paper "*A quality model for conceptual models of MDD environments*," the authors, B. Marín et al. present a quality model, which not only encapsulates defect types that are related to conceptual models, but also takes advantage of current standards in order to automate defect detection in MDD environments. The detection of defects is a promising technique to evaluate software quality, which is emerging as a suitable alternative for MDD processes.

In "*An empirical study of social networks metrics in object oriented software*," G. Concas et al. study the application to object-oriented software of new metrics, derived from Social Network Analysis. These metrics are compared with other traditional software metrics, like the Chidamber-Kemerer suite, and software graph metrics. They found that the empirical distributions systematically show fat-tails for all the metrics. These features appear to be typical properties of these software metrics.

In the paper "*Exploring the eradication of code smells: an empirical and theoretical perspective*," S. Counsell et al. present three studies about Code Smells. Key findings of the study were that first, smells requiring application of simple refactoring interventions were eradicated in favor of smells requiring more complex refactoring interventions; second, a wide range of conflicts and anomalies soon emerged when trying to identify smelly code; an interesting result with respect to comment lines was also observed. Finally, the perceived effort to eradicate a smell may be a key factor in explaining why smell eradication is avoided by developers.

In the last paper "On the *use of issue tracking annotations for improving developer activity metrics,*" A. Meneely and L. A. Williams propose two annotations to enrich the measurement of collaboration among developers, and to be used in the issue of systems tracking. These annotations record the originator of the solution and the approver of the solution. They examined the online discussions about 602 issues concerning the OpenMRS healthcare web application. Applying social network analysis to the data, they found that approvers are likely to score high in measures of centrality and hierarchical clustering computed on the developers' social network.

We hope you will find this issue stimulating and useful and look forward for your evaluable feedback.

*Giulio Concas*
*Giovanni Cantone*
*Ewan Tempero*
*Hongyu Zhang*

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 685950, 17 pages
doi:10.1155/2010/685950

*Research Article*

# The Economics of Community Open Source Software Projects: An Empirical Analysis of Maintenance Effort

**Eugenio Capra, Chiara Francalanci, and Francesco Merlo**

*Dipartimento di Elettronica ed Informazione, Politecnico di Milano, via Ponzio 34/5, 20133 Milano, Italy*

Correspondence should be addressed to Francesco Merlo, merlo@elet.polimi.it

Received 20 November 2009; Accepted 12 July 2010

Academic Editor: Giulio Concas



Previous contributions in the empirical software engineering literature have consistently observed a quality degradation effect of proprietary code as a consequence of maintenance. This degradation effect, referred to as entropy effect, has been recognized to be responsible for significant increases in maintenance effort. In the Open Source context, the quality of code is a fundamental design principle. As a consequence, the maintenance effort of Open Source applications may not show a similar increasing trend over time. The goal of this paper is to empirically verify the entropy effect for a sample of 4,289 community Open Source application versions. Analyses are based on the comparison with an estimate of effort obtained with a traditional effort estimation model. Findings indicate that community Open Source applications show a slower growth of maintenance effort over time, and, therefore, are less subject to the entropy effect.

## 1. Introduction

Authors in the software economics field concur that there exists a tradeoff between development and maintenance costs [1–3]. If development budgets are tight, subsequent maintenance operations will be more costly. Understanding, modeling, and empirically verifying this tradeoff have always been important research issues in the empirical software engineering literature.

The cost efficiency of maintenance operations is affected by many factors. A fundamental cost driver is the quality of code [3]. A quality degradation effect of code has been observed as a consequence of maintenance operations [4, 5]. Maintenance operations have been found to increase coupling, reduce modularity, create a gap between actual code and documentation and, more generally, make code more "chaotic". Consequently, the cost of maintenance operations tends to grow over time.

This tradeoff between quality and costs is broadly analyzed in the literature for proprietary software. A commonly accepted variable measuring the overall degradation of software quality is *entropy*. Entropy represents a powerful theoretical variable that aggregates several aspects of quality degradation to support the assessment of their global effect on overall attributes of a software application, primarily maintainability and costs. From an operating standpoint, entropy is usually defined based upon the structural properties of code, such as method calls between different classes, shared objects, coupling, and modularity. Overall, the literature focuses on the effect of entropy on maintainability from a technical perspective, while fewer works investigate the impact of entropy on costs. However, research contributions concur on the chain of causal relationships among entropy, maintainability, and maintenance costs [5].

This chain of causal relationships is challenged by Open Source (OS) development and maintenance practices. In the OS context, the quality of code is a fundamental design principle [6, 7]. OS developers are strongly motivated and skilled, tend to view software as an artifact rather than a product, are willing to modify code for the sake of experimentation or to achieve ambitious performance objectives, and often consider the quality of their implementation effort as an intellectual reward [8]. As a consequence, the quality degradation effect measured by entropy cannot be assumed to be valid. Furthermore, the virtual coordination practices among geographically distributed OS developers can benefit from higher levels of code quality. Virtual coordination is recognized to be more cumbersome than

face-to-face meetings and a more modular, well-structured, and comprehensible code has been found to reduce the coordination effort [9, 10]. From this perspective, higher quality of code is a coordination requirement and, as a consequence, the entropy effect may be further reduced.

The goal of this paper is to empirically analyze the entropy effect for a sample of community OS applications. The paper posits that community OS applications have higher average code quality and, therefore, are less subject to entropy. Consequently, community OS applications are hypothesized to require a lower maintenance effort and a lower need for refactoring operations aimed at restoring quality. In order to verify these hypotheses, the paper proposes a new operational definition of entropy, referred to as *time-entropy*, that is, based on the concept of autocorrelation of the maintenance effort variable. The main novelty of this metric is that it is independent of code. This paper's measure of maintenance effort is also independent of code, as it represents a direct measure of the time spent in maintenance activities. For the sake of simplicity, in the following the terms *costs* and *effort* will be used as synonyms to indicate *maintenance effort*. The paper verifies whether community OS applications are more or less entropic than proprietary applications by comparing the empirical measure of time entropy for a sample of community OS applications with the corresponding estimate of time entropy obtained with a traditional cost model designed for proprietary software.

The presentation is organized as follows. Section 2 presents an overview of existing cost models in the software engineering literature. Section 3 presents our research hypotheses. Section 4 discusses the operating definitions of effort, time entropy, and refactoring. Section 5 presents results, which are then discussed in Section 6. Finally, threats to validity and future work are discussed in Section 7.

## 2. Related Work

*2.1. General Cost Models.* The estimation of software development costs and the economic analysis of software maintenance have always been important research subjects in the software engineering literature, since the early works of Boehm [2] and Lehman [1]. Several prediction models and techniques have been proposed (e.g., [11–13]) as well as comprehensive evaluation methodologies (e.g., Kemerer [14] and Briand et al. [15]). The literature makes a distinction between the initial development cost and the cost of subsequent maintenance activities. At a given time point, the total development cost of an application is defined as the sum of the initial development cost and the cost of all subsequent maintenance activities [3].

The Constructive Cost Model (CoCoMo) has been defined by Boehm in the early 80's [16] as the first nonproprietary cost estimation model. CoCoMo provides a framework to calculate initial development costs based on an estimate of the time and effort (person-months) required to develop a target number of lines of code (SLOC). The functional characteristics of an application are taken into account by means of context-dependent parameters. The model has continuously evolved over time: Ada CoCoMo (released in 1987) and CoCoMo 2.0 (released in 1995) represent the main developments of the original CoCoMo (see [17]). The latter takes into account requirements, in addition to SLOC. Requirements are measured in function points (FPs), that is, the number of elementary operations performed by a software application [18–20]. CoCoMo's basic estimation model is the following:

$$E = a \cdot \mathrm{SIZE}^b, \tag{1}$$

where SIZE represents an estimate of the SLOC or FPs of an application, $a$ and $b$ are context-dependent parameters, and $E$ is the development effort, usually measured in person-months. Even though CoCoMo does not focus on maintenance, the model has been extended (cf. [21]) to estimate maintenance costs by means of a scaling factor. Such factor, called Annual Change of Traffic (ACT), is an estimate of the average size of changes (measured in changed SLOC) that are made to the application over a given maintenance period.

Maintenance operations have been empirically found to account for about 75% of the total development cost of an application over the entire application's life cycle [13, 22, 23]. The concept of entropy has been proposed in the literature to support the estimate of maintenance costs. Entropy is defined as a measure of the internal complexity of an application's code [24] and of the quantity of the information embedded within the code itself [25]. Bianchi et al. [4] point to entropy as a primary cause for software quality degradation. According to Bianchi et al. [4], maintenance operations decrease software modularity and increase internal coupling, thus making code more complex and increasing the quantity of information, that is, the effort required to understand and manipulate code.

Tan and Mookerjee [5] propose a model supporting the estimate of total development costs based on the concept of entropy. Similar to CoCoMo 2.0, Tan and Mookerjee's model measures the size of applications in function points. The cost of a maintenance operation is expressed as

$$C = \gamma_0 + \gamma_1 M_i + \gamma_2 M_i^2 + \gamma_3 M_i M_j e^{\kappa n}, \tag{2}$$

where

(i) $\gamma_0$ is the fixed cost of maintenance, defined as the effort required to plan and organize a maintenance initiative;

(ii) $\gamma_1 M_i$ is the linear cost of specifying requirements and developing new functionalities;

(iii) $\gamma_2 M_i^2$ is the quadratic cost associated with the integration of new modules among themselves, consistent with previous studies on software integration [23];

(iv) $\gamma_3 M_i M_j e^{\kappa n}$ is the cost associated with the integration of new modules with preexisting software (composed of $M_j$ modules), which grows exponentially with the number $n$ of previous maintenance operations and consequent entropy; the parameter $\kappa$ is used to model the growth rate of system degradation and is directly related to the entropy of the application.

Initial implementation costs can be estimated as the cost of a maintenance operation that does not require any integration effort with preexisting code, that is, by excluding the fourth component of costs from Expression (2). At a given time point, total development costs can be obtained as the summation of the cost of all maintenance operations. Due to the exponential impact of entropy on maintenance costs, there exists a maximum number of maintenance operations above which reimplementation becomes less costly than maintenance. The model presented by Tan and Mookerjee [5] represents the most complete state-of-the-art cost estimation model that accounts for both initial development and subsequent maintenance costs. In Section 5, the model is used to estimate the total development and maintenance effort of a reference proprietary application to be compared with our empirical measure of OS maintenance effort.

It should be noted that a limitation of Tan and Mookerjee's model [5] is that it fails to consider refactoring as an alternative to reimplementation. Refactoring is defined by Fowler et al. [26] as

> "*The process of changing an application without altering its external behavior, but improving its internal structure*".

Mens and Tourwé [27] provide an extensive overview of existing research in the field of software refactoring. They observe that refactoring is a less drastic and sometimes more viable alternative to replacement. They also discuss refactoring activities, techniques, and formalisms. A quantitative model to evaluate the impact of refactoring on software maintainability is proposed by Kataoka et al. [28]. Software maintainability is described by means of four metrics: coupling, cohesion, size and complexity of modules, and appropriateness of naming rules. The impact of refactoring is evaluated by empirically measuring maintainability before and after a refactoring operation. The relationship between refactoring and maintainability is also empirically analyzed by Fowler et al. [26]. Entropy is seen as the primary obstacle to maintainability and, hence, a fundamental driver of maintenance costs. However, a model to estimate the impact of refactoring on entropy and costs is not provided.

Chan et al. [29] discuss how a volatile user environment causes software maintainability to deteriorate with the age of an application. Refactoring and replacement are recommended to improve maintainability. A quantitative model to optimize the timing of these operations is also proposed. The model takes into consideration the user environment, the effectiveness of rewriting, the technology platform, the quality of development, and the familiarity of developers with existing software. Entropy is associated not only with the complexity of code, but also with a misalignment between the application and its user environment. While the model supports the scheduling of refactoring operations to reduce entropy, the cost impact of refactoring is not analyzed.

*2.2. Open Source Effort Estimation.* The models discussed in the previous section have been designed for and tested on proprietary software projects. They assume a direct impact of entropy on maintainability and, hence, on maintenance costs. Testing these relationships on OS applications is still an open research problem [9, 30]. Amor et al. [31] argue that traditional cost/effort estimation models can be enhanced in the OS context by taking advantage of additional information extracted from a variety of open information sources, such as source code management systems, mailing lists, and bug tracking systems. Yu [32] notes that in the OS context the prediction of maintenance costs should be based on indirect measures of effort, since actual effort is rarely documented, contrary to the work practices of proprietary software projects.

Open Source was born as a development practice to share code among freelance developers and academics [33]. However, OS is currently becoming a business model [34–36]. As a consequence, research providing empirical evidence of the cost benefits of OS practices has been repeatedly called for [7, 35]. The existing literature on OS costs tends to take the user perspective and focus on the comparison of different acquisition cost strategies related to different types of licenses [34, 37]. As noted by Riehle [38], there is a lack of knowledge on the long-term consequences of OS adoption, especially from an economic perspective.

Open Source supporters claim that OS practices increase the quality of the software artifact [6, 39]. Higher software quality should translate into greater maintainability. As discussed in the previous section, the literature on proprietary software has identified entropy as a primary cause for lower maintainability. A higher-quality software artifact should be less entropic and, hence, involve lower maintenance effort. While the acquisition cost convenience of OS is broadly advocated as a fundamental advantage over proprietary software [36, 40, 41], only a few studies provide empirical evidence of the economic benefits of OS over an application's life cycle (see, e.g, [7]).

Moreover, it should be noted that, although OS development cannot be identified as an agile development methodology, significant similarities have been found in several areas [42]. In fact, both OS practices and agile methods promote the focus on individuals and "artisan" developers rather than structures and processes, the self-organization of teams, a close interaction with users, early delivering of working code, good design, and simplicity. On the other hand, agile methods and OS differ in at least two significant aspects. First, whereas agile methods are reported to be successful only for small team (about 15–20 persons) [43], there are successful OS communities with hundreds of developers spread around the world. Second, agile methods exalt close and personal contact within the development team, defining specific practices such as pair programming and daily meetings. This is rarely possible in OS communities, as developers usually live in different locations and mainly communicate by e-mails, IRC channels, and other virtual tools. In agile methodologies there is no clear distinction between maintenance and development, and refactoring tend to be a continuous process. As OS is very similar to agile methods, this peculiarity should be taken into account when analysing OS development costs.

*2.3. Entropy.* In Tan and Mookerjee's model, the cost effect of the quality degradation of software systems over their life cycle is represented by the exponential term in (2). Since entropy is a parameter of this exponential term, the model implies that the cost impact of quality degradation is more significant when entropy is higher. Tan and Mookerjee's model does not provide an operating definition of entropy, but refers to previous literature, as discussed below.

It is widely accepted that the maintainability of a software system is affected by quality and, more specifically, by a subset of quality attributes measuring the complexity of code [4, 44, 45]. A number of different metrics have been proposed to measure quality and complexity (Zuse [46] provides a comprehensive review of these metrics). However, these metrics are often difficult to choose and interpret, as they focus on specific and partially overlapping aspects of code characteristics [47]. Bianchi et al. [4] have defined entropy as a subset of complexity metrics that assess the degree of chaos in a software system's traceability—that is, the ability to trace back the current structure of code to the history of changes, which must be documented for all the components of a software system, at different abstraction levels, and for all the mutual relationships among components. Tan and Mookerjee refer to this definition of entropy in their model [5].

The definition of entropy provided by Bianchi et al. [4] is strictly related to the code structure of an application. Measuring their metrics involves a significant code analysis effort. From a theoretical standpoint, such measures of entropy represent the aggregate effect of a number of variables: a more direct measure of entropy, possibly not based on the indirect measurement of its causal variables, could provide interesting insights. An alternative measure of entropy that goes beyond code structure has been proposed by Hassan and Holt [48]. They observe that code-based measures do not quantify the actual complexity faced by developers and by project managers. Their measure of entropy focuses on the effect of code complexity as the input to a development and management process that must deliver an outcome. This new concept of entropy originates from the historical definition of entropy provided by Shannon and Weaver [49]. Shannon's entropy measures the uncertainty of information, which determines the minimum number of bits required to uniquely distinguish a distribution of data, and ultimately defines the best possible compression for the distribution (i.e., the output of the system). Hassan and Holt [48] view the development process of a software system as a process that produces information, interpret such information as the modifications made to the source code files, and define entropy as the amount of uncertainty in the development process. Intuitively, if all the files of a software system are modified, developers and managers will face a more complex process to keep track of all modifications. The number of bits needed to remember all these modifications is higher than the number of bits required to describe only a limited number of modifications. With this definition, entropy can be measured by examining the logs of the repository of a software project.

Hassan and Holt [48] have applied their definition of entropy to a number of OS projects. However, the impact of entropy on maintenance effort is not discussed. Hassan and Holt's entropy is limited to considering the number of files that are modified, but does not consider the impact of each modification and their mutual relationships over time.

In this paper, we draw from Hassan and Holt [48] the general definition of entropy as the overall effect on the development process of several code complexity variables. In compliance with this definition, in Section 4.4 we propose a new metric of entropy, referred to as *time entropy*, that is, not code-based and that takes into account the time dimension of the development process. As discussed in the previous sections, the time dimension is fundamental to understand the cumulative effect of maintenance operations and their ultimate impact on costs.

## 3. Research Hypotheses

There exist a number of differences between traditional closed source and OS projects, especially when considering community OS development practices. First of all, OS developers are more motivated towards quality. Although in some cases OS is currently becoming a business model, it was born from sharing and cooperation ideals (as stated by the "philosophy" of the Free Software Movement, [50], and the GNU Manifesto, [33]). For most OS community developers, it represents a philosophy close to their personal beliefs [8]. Social objectives such as the elimination of the digital divide and software accessibility to individuals and smaller companies constitute fundamental motivational drivers for many OS developers. This coherence between job culture and personal beliefs makes OS developers particularly motivated towards the quality of their software artifact. In turn, this can reduce the entropy effect discussed in the previous sections. The implementation schedule of voluntary developers in OS communities is also less tight [31]. In many community OS projects, coordination is looser and developers set their own deadlines according to their commitment to the project, which is most often part time [51]. In contrast, the implementation deadlines of proprietary applications are revenue driven. Delays involve direct economic losses, such as penalties, as well as indirect opportunity costs. This financial pressure leads developers to time-oriented implementation choices that negatively affect quality, as per the entropy principle.

Finally, OS developers are often geographically distributed and may be employed by separate companies [37]. Their distribution across geographical regions and companies reduces their ability to physically meet to discuss design issues and cooperate in problem solving. Even though traditional closed source projects may be developed by groups of employees working in different locations, corporate structures tend to make communication and coordination more frequent and effective [52]. In OS projects, virtual coordination practices and tools are largely applied [53, 54]. However, it has been observed that virtual coordination is more cumbersome and less efficient, especially for decision-making activities [55]. A more modular, understandable,

and well-designed code can greatly help virtual coordination. The structure of code represents a fundamental aid to coordination, as it helps designers to allocate jobs, locate changes made by other designers and developers, understand the effect of those changes, and integrate new contributions [9, 56]. A recent stream of literature states that a lower degree of entropy is necessary to enable the cooperation among distributed and looser groups of developers [9, 57].

More successful and higher-quality projects are likely to have a lower degree of entropy, regardless of the openness of the code. However, there are such differences between traditional closed source and community OS projects, that lead to suppose there may be a the average entropy level may be significantly different according to the development and governance style. Although these considerations do not represent a sufficient argument to state that the entropy effect does not exist in community OS projects, they suggest a lower entropy effect. This leads us to our first research hypothesis.

(H1) Community Open Source applications are less entropic than proprietary applications, that is, their quality degrades more slowly over time.

The quality of proprietary applications is periodically restored through refactorings [28, 29]. By restoring quality, a refactoring reduces entropy and, hence, the required maintenance effort. Fowler notes that refactoring is performed with two different objectives [26]: (a) to increase software quality and (b) to integrate functionalities developed by different developers or added to different versions of the same application. This second driver of refactoring should be predominant in an OS context, especially when considering communities of developers. We already noted in Section 2 that OS development practices are very similar to agile methods. Agile methods encourage frequent code releases, as they aim at shifting the focus from processes to individuals and code [58]. As a consequence, refactoring becomes a way for making the application evolve, tends to be continuous, and to have a different impact than in traditional applications. In OS contexts refactorings are not aimed at restoring quality only and, therefore, are independent of quality degradation, that is, of the entropy effect. More likely, refactorings may be a way to integrate different contributions and merge them into new releases of the application. This leads to our second research hypothesis.

(H2) For community Open Source applications, entropy and the frequency of refactoring are not correlated.

An accepted result in the software engineering literature is that maintenance effort is lower if the quality of code is higher [3]. Therefore, if applications are less entropic, maintenance effort should also be lower. Although this relationship between entropy and maintenance effort is consolidated for traditional proprietary applications, it has never been empirically tested for OS applications. Our third research hypothesis is aimed at verifying the relationship between entropy and maintenance effort in the OS context. We hypothesize in (H1) that community OS applications are less subject to entropy growth than proprietary applications. Hypothesis (H3) is aimed at verifying whether, although lower, the entropy effect is still responsible for inefficiencies even in the OS context.

(H3) In a community Open Source context, the maintenance effort of less entropic applications is lower than the maintenance effort of more entropic applications.

Overall, testing hypotheses (H1) and (H2) helps verify that community OS applications are less subject to the entropy effect compared to proprietary applications and, consequently, refactoring is not aimed at reducing entropy. Testing (H3) empirically verifies whether entropy remains a cost driver in an OS community of developers. Furthermore, by verifying hypothesis (H3), we reinforce our variables, metrics, and results, by testing whether they are consistent with consolidated principles of the software engineering literature. Please note that hypothesis (H3) is not tautologic. Entropy is linked with the autocorrelation of cost over time, and measures to what extent current costs are affected by costs in the past. This is intrinsically different from the average value of cost.

## 4. Variable Definition and Operationalization

This section presents the operationalization of the variables involved in testing our research hypotheses. The metrics that are used in the operationalization of other variables are discussed first.

### 4.1. Software Size Metrics

*Functionalities.* A functionality is defined as an element of the Graphical User Interface (GUI) that can be activated by users. Sample functionalities are menu items, command buttons, and toolbar keys. Referring to a generic application $k$, the functionality set $\mathcal{F}_k(i)$ of version $i$ is defined as its set of functionalities. The number of functionalities $F_k(i)$ is defined as the cardinality of the set $\mathcal{F}_k(i)$:

$$F_k(i) = |\mathcal{F}_k(i)|. \tag{3}$$

If $\mathcal{F}_k(i)$ and $\mathcal{F}_k(j)$ are the set of functionalities of versions $i$ and $j$, respectively, the variation of functionalities $\Delta F_k(i, j)$ between versions $i$ and $j$ is defined as the cardinality of the symmetric set difference between $\mathcal{F}_k(i)$ and $\mathcal{F}_k(j)$:

$$\Delta F_k(i, j) = |\mathcal{F}_k(i) \ominus \mathcal{F}_k(j)|. \tag{4}$$

The symmetric set difference between two sets $A$ and $B$ is defined as the set of elements belonging to one but not both sets, and is commonly written as $A \ominus B$. In other words, it is the union of the complement of $A$ with respect to $B$ and of $B$ with respect to $A$, and corresponds to the XOR operator in Boolean logic.

*Methods.* The set of methods of an application version is composed of the methods of all its classes. The number of methods comprises all methods, that is, it includes public,

protected, private and package-only methods. Considering a generic application $k$, the number of methods $M_k(i)$ of version $i$ is defined as the cardinality of the corresponding set of methods $\mathcal{M}_k(i)$:

$$M_k(i) = |\mathcal{M}_k(i)|. \tag{5}$$

If $\mathcal{M}_k(i)$ and $\mathcal{M}_k(j)$ are the set of methods of versions $i$ and $j$, respectively, the variation of methods $\Delta M_k(i, j)$ between versions $i$ and $j$ is defined as the cardinality of the symmetric set difference between $\mathcal{M}_k(i)$ and $\mathcal{M}_k(j)$:

$$\Delta M_k(i, j) = |\mathcal{M}_k(i) \ominus \mathcal{M}_k(j)|. \tag{6}$$

*SLOC.* The number of source lines of code (SLOC) is defined according to the basic definition of physical line of code provided by Park [59]. A physical SLOC has been defined as a line ending in a newline or end-of-file marker, and which contains at least one nonwhitespace noncomment character. Comment delimiters (characters other than newlines starting and ending a comment) are considered comment characters. Data lines only including white spaces (e.g., lines with only tabs and spaces in multi line strings) are not included. The variation of SLOC between two versions $i$ and $j$ of a generic application $k$ is defined as

$$\Delta\mathrm{SLOC}_k(i, j) = \mathrm{SLOC}_k(i) - \mathrm{SLOC}_k(j). \tag{7}$$

For application version $i$, the number of SLOC per method is defined as

$$\mathrm{SLOC}_k^M(i) = \frac{\mathrm{SLOC}_k(i)}{M_k(i)}. \tag{8}$$

*4.2. Maintenance Effort.* Development effort is measured in person-days. The development time $t_k(i)$ of version $i$ of application $k$ is defined as the number of days elapsed between the release of version $i$ and the release of previous version $i-1$. Development effort $E_k(i)$ measured in person-days is obtained by multiplying development time $t_k(i)$ by the following correction factors:

(i) $n_k(i)$, the number of project administrators or developers;

(ii) $\alpha_k$, the average fraction of time that each contributor (administrator or developer) devotes to the project;

(iii) $\beta_k$, the percentage of active administrators and developers in the project team.

The complete expression for maintenance effort is

$$E_k(i) = t_k(i) \cdot \left[\alpha_k^{\mathrm{admin}} \cdot \beta_k^{\mathrm{admin}} \cdot n_k^{\mathrm{admin}}(i) + \alpha_k^{\mathrm{devel}} \cdot \beta_k^{\mathrm{devel}} \cdot n_k^{\mathrm{devel}}(i)\right]. \tag{9}$$

The unit maintenance effort $e_k(i)$ is defined as the ratio of maintenance effort (cf. Expression (9)) to the variation of methods from the previous version (cf. Expression (6)):

$$e_k(i) = \frac{E_k(i)}{\Delta M_k(i, i-1)}. \tag{10}$$

*4.3. Refactoring.* A refactoring operation is defined by Fowler et al. [26] as "*a change made in the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*". To operationalize this definition, we have considered the variation of functionalities $\Delta F_k(i, i-1)$ (which can be assumed as a proxy for measuring changes in the "observable behavior" of an application) and the variation of source lines of code $\Delta\mathrm{SLOC}_k(i, i-1)$. Table 1 provides an overview of the different types of operations that can be identified by considering different combinations of the values of the two variables $\Delta F_k(i, i-1)$ and $\Delta\mathrm{SLOC}_k(i, i-1)$.

Consistent with Fowler's definition, case A in Table 1 can be classified as a refactoring operation since there is no variation of functionalities and the size of source code is reduced. In case B, the number of functionalities grows, but the size of the source code decreases. This decrease suggests that code has been reorganized and new functionalities are likely to correspond to a reorganization of the interface rather than the implementation of new requirements. Therefore, refactoring can be considered predominant over development. In case C, a growth of the size of source code is accompanied by no variation of functionalities. This typically occurs when developers add the logic behind a graphical user interface that was previously mocked up and, as a consequence, development can be considered predominant. Finally, case D identifies a "classic" development operation, since functionalities are added and the source code grows accordingly. Versions classified as refactorings (cases A and B) have been manually verified by inspecting changelogs and documentation. We could not find sufficient information for 71 versions due to missing changelogs and scarce documentation. The other versions were found to be correctly classified by our metric. Note that cases involving negative variations of functionalities (labeled with $X$ in Table 1) have not been addressed since no occurrences were found in our sample.

We define as refactoring all the application versions that belong to cases A and B. Formally, the occurrence of a refactoring operation in version $i$ of application $k$ is measured by the boolean variable $\mathrm{ref}_k(i)$ which evaluates to 1 if a refactoring occurs, to 0 otherwise:

$$\mathrm{ref}_k(i) = \begin{cases} 1, & \text{if } \Delta F_k(i, i-1) \geq 0 \wedge \Delta\mathrm{SLOC}_k(i, i-1) \leq 0, \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

With this definition, $\mathrm{ref}_k(i)$ indicates that a refactoring operation is performed in version $i$ if the variation of functionalities (cf. Expression (4)) is greater than or equal to 0, and the variation of source lines of code (cf. Expression (7)) with respect to the previous version is smaller than or equal to 0.

The frequency of refactoring $f_k^{\mathrm{ref}}$ is defined as the ratio of the number of refactoring operations $N_k^{\mathrm{ref}}$ to the total number of versions $V_k$ of application $k$:

$$f_k^{\mathrm{ref}} = \frac{N_k^{\mathrm{ref}}}{V_k}, \tag{12}$$

Table 1: Types of operations on source code.

| | | $\Delta F_k(i, i-1)$ | | |
|---|---|---|---|---|
| | | $< 0$ | $= 0$ | $> 0$ |
| $\Delta \mathrm{SLOC}_k(i, i-1)$ | $\leq 0$ | [X] interface rationalization | [A] refactoring | [B] predominantly refactoring |
| | $> 0$ | [X] interface rationalization | [C] predominantly development | [D] development |

where the number of refactorings $N_k^{\mathrm{ref}}$ is defined as follows:

$$N_k^{\mathrm{ref}} = \sum_{i=0}^{V_k} \mathrm{ref}_k(i). \tag{13}$$

*4.4. Entropy.* The literature concurs that entropy causes an increase in maintenance costs over time. If entropy is high, unit maintenance costs calculated for subsequent versions should grow. Several metrics of entropy have been proposed in the literature, as described in Section 2.3. Most of these metrics focus on code structure. As noted in Section 2.3, entropy represents the aggregate effect of a number of variables, but a direct measure of entropy, that is, not based on the indirect measure of code-based variables is still missing.

We propose a new definition of entropy, which we name *time entropy*, that aims at encompassing all the variables that, over time, cause software degradation as a consequence of multiple subsequent maintenance operations. If an application is viewed as a dynamic system, entropy can be considered as the memory of the system itself. This memory affects the effort variable, which, in turn, should be autocorrelated over time. Consequently, we measure time entropy as the autocorrelation of the values of unit maintenance effort, calculated for subsequent versions of the same application. For a detailed discussion of the autocorrelation function, please refer to Papoulis and Pillai [60].

Let us consider a generic application $k$ that evolves over time, leading to a sequence of versions $v_0, \ldots, v_{V_k-1}$, where $V_k$ is the total number of versions as defined in Section 4.3. For each version $i$ of application $k$, $e_k(i)$ has been defined as the unit maintenance effort (cf. Expression (10)). The autocorrelation $a_i(\tau)$ of unit maintenance effort for version $i$ at lag $\tau$ is defined as

$$a_i(\tau) = e_k(i) \cdot e_k(i-\tau). \tag{14}$$

Considering that, for each lag $\tau$, application $k$ consists of the set of all the versions $V_0, \ldots, V_\tau$, the average autocorrelation of application $k$ at lag $\tau$ is defined as:

$$A_k(\tau) = \frac{1}{v_\tau - \tau + 1} \cdot \sum_{i=0}^{\tau} a_i(\tau) \quad \text{with } 0 \geq \tau < V_k. \tag{15}$$

The average time entropy $S_k$ of application $k$ is then calculated as

$$S_k = \frac{1}{V_k} \cdot \sum_{\tau=0}^{V_k-1} A_k(\tau). \tag{16}$$

Expression (16) provides the definition to compute the average time entropy of application $k$ by considering its whole version history. However, it can be generalized in order to compute the time entropy value by considering only a part of the evolution history: the average time entropy of application $k$ considering only the last $n$ versions (i.e., version $v_{V_k-n+1}, \ldots, v_{V_k}$ with $n < V_k$) is defined as

$$S_k(n) = \frac{1}{n+1} \cdot \sum_{\tau=0}^{n} A_k(\tau). \tag{17}$$

Given the definition of time entropy, we define the time-entropy variation $\Delta S_k(j, i)$ between two different versions $i$ and $j$ of application $k$ as

$$\Delta S_k(j, i) = S_k(j) - S_k(i) \quad \text{with } j > i, \tag{18}$$

and the relative time-entropy variation $\Delta^{\%} S_k(j, i)$ of version $j$ with respect to version $i$ as

$$\Delta^{\%} S_k(j, i) = \frac{\Delta S_k(j, i)}{S_k(i)} = \frac{S_k(j) - S_k(i)}{S_k(i)} \quad \text{with } j > i. \tag{19}$$

Our metric is labelled *time entropy*, since it focuses on the temporal growth of entropy, rather than its absolute value. From a theoretical standpoint, our metric is tightly related to the concept of entropy defined in previous software engineering literature. In particular, we draw from Hassan and Holt [48] the definition of entropy as the overall effect on the development process of several code complexity variables. Section 2 thoroughly discusses how the entropy of software applications tends to grow over time as a consequence of maintenance. The software engineering literature provides strong evidence of this temporal evolution of entropy. The growth of entropy is also indicated as a fundamental cause for application replacement and refactoring, which involve significant investments and represent a fundamental concern in numerous previous papers [4, 5, 23, 61]. For these reasons, our metric of entropy is based on the assumption that the time evolution of entropy is a strong and measurable phenomenon.

Figure 1 provides an example to illustrate the definitions provided in this section. Application $k$ has four different versions, labeled $V_0, \ldots, V_3$. The initial development effort is identified by $e(0)$, and each subsequent maintenance effort $e(1), \ldots, e(3)$ is associated with the corresponding one. Expression (14) let us compute the autocorrelation $a_i(\tau)$ of maintenance effort for each version, considering

Table 2: Summary statistics of our application sample (total of 393 applications, corresponding to 4,289 versions).

| Variable | Symbol | Mean | Median | Min | Max | St.Dev. |
|---|---|---|---|---|---|---|
| Administrators | | 2.13 | 2.00 | 1.00 | 15.00 | 1.68 |
| Developers | | 4.38 | 1.00 | 0.00 | 132.00 | 10.31 |
| Team | | 6.52 | 3.00 | 1.00 | 141.00 | 11.24 |
| Versions | $V$ | 10.91 | 7.00 | 2.00 | 157.00 | 14.37 |
| Methods | $M$ | 1,398.00 | 579.00 | 30.00 | 32,110.00 | 2,692.00 |
| Time-Entropy | $S$ | 265.29 | 0.24 | 4.43E-5 | 65,783.00 | 3,409.80 |
| Unit Effort | $e$ | 7.09 | 0.26 | 4.55E-3 | 1,737.00 | 88.90 |
| *SourceForge.net* Ranking | | 16,354.00 | 8,349.00 | 25.00 | 86,840.00 | 19,406.00 |
| Review Rate (months/version) | | 7.04 | 4.42 | 0.05 | 40.15 | 7.19 |

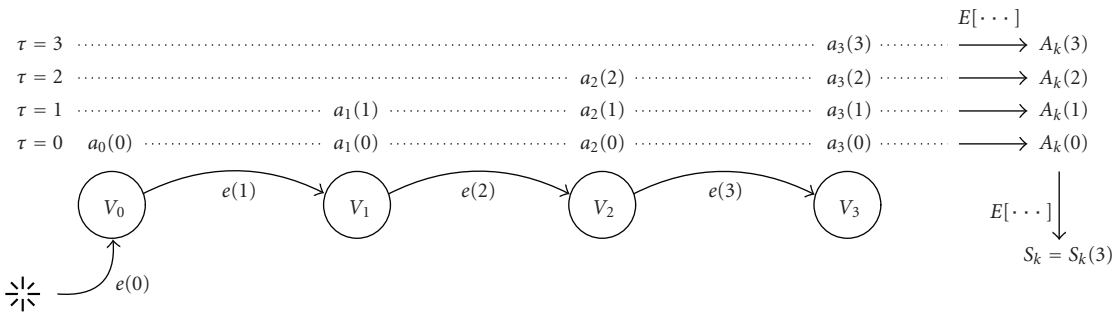


Figure 1: An example describing the computation of the time entropy $S_k$ of a generic application with four versions.

different time lags. In Figure 1, $a_i(\tau)$ has been computed for all versions by considering all possible time lags, that is, $\tau$ ranging from 0 to 3. By calculating the mean value of all $a_i(\tau)$ for different values of $\tau$ (cf. Expression (15)), we calculate the average autocorrelation $A_k(\tau)$ of application $k$ for each time lag $\tau$. Finally, Expression (16) allows us to compute the actual time entropy $S_k$ of application $k$ over its whole version history, as the mean value of $A_k(\tau)$, for $\tau = 0, \ldots, 3$. In this case, $S_k = S_k(3)$, since the sample application has four versions.

Our metric measures the extent to which the maintenance effort incurred in the past (i.e., $e_k(i - \tau)$) affects the maintenance effort, that is, incurred for the current version (i.e., $e_k(i)$). In turn, this depends on the extent to which the previous history of maintenance operations has negatively affected the quality of the system. The autocorrelation function takes into consideration the effect of maintenance operations carried out both on recent versions (low values of $\tau$) and on the older versions (high values of $\tau$). Time entropy $S_k$ is then calculated as a mean value over $\tau$. Note that time entropy could not be measured by means of unit effort, since it may grow even if the mean values of unit effort across the different versions of an application are not an increasing monotonic function over time. This is due to a possible varying frequency of refactoring across applications. On the contrary, autocorrelation takes into account the varying frequency of refactoring by means of the time lag $\tau$. When $\tau$ is lower than the time between two subsequent refactorings, autocorrelation is not affected by refactoring operations and it grows when unit effort grows between subsequent refactorings.

## 5. Methodology and Results

This section presents our sample (Section 5.1), data analysis tool (Section 5.2), and empirical findings (Section 5.3).

*5.1. Data Sample.* Data have been collected by mining the *SourceForge.net* repository, one of the most referenced and active online repositories of community OS projects. Mining online repositories such as *SourceForge.net* can lead to controversial results because of the varying quality of available data [62]. The following criteria have been applied to guarantee the reliability of our data set

(i) Programming Language: Java;

(ii) Project Maturity: beta status or higher. Less mature applications have been excluded because of their instability and low significance;

(iii) Version History: at least 5 versions released;

(iv) Graphic User Interface (GUI), where present: Java Swing, AWT or SWT.

These criteria have led us to a first data set including 1,411 applications. A further refinement step has been performed to remove void versions with no methods and corresponding void applications with no versions. This second filtering stage provided a final selection of 393 applications, corresponding to 4,289 versions (online Appendix B reports the full list of applications). Table 2 presents the summary statistics of our sample of applications.

For the final application sample, the number of project administrators and developers that we have used is the one

Table 3: Summary statistics from the survey on maintenance effort.

| Team Member Class | $n$ total | $n$ active | Average (hr/week) | St. Dev. | Conf. Int. ($\alpha/2 = 0.025$) | $\overline{\alpha}$ | $\overline{\beta}$ |
|---|---|---|---|---|---|---|---|
| Administrator | 156 | 111 | 6.21 | ±7.40 | ±0.21 | 0.1553 | 0.7115 |
| Developer | 497 | 313 | 6.63 | ±9.65 | ±0.13 | 0.1658 | 0.6298 |
| Global | 653 | 424 | 6.41 | ±8.45 | ±0.11 | 0.1603 | 0.6493 |

officially listed by *SourceForge.net* in the home page of each project. Factors $\alpha_k$ and $\beta_k$ of our maintenance effort metric (Section 4.2) have been empirically estimated by surveying 2,564 OS project administrators and developers involved in the projects of our sample. The questionnaire has been sent to individual contributors in order to gather the actual time devoted by each respondent to each project he or she was involved in. We have received a total of 653 answers, with a global response ratio of 25.5%. Online Appendix C reports the complete questionnaire that has been used for the survey.

As for all empirical research works based on surveys, the accuracy of the data that we have collected may be a threat to the validity of our work. In order to mitigate this risk, we have performed the following verifications. First, the respondents of the survey are a subset of our sample and thus selected according to the same criteria of significance. Second, our survey includes only structured and quantitative questions, in order to avoid the need for subsequent interpretations of responses. Third, all answers have been checked for consistency and missing values. Whenever inconsistencies have been found, respondents have been contacted by e-mail to clear doubts or exclude them from the survey.

Based on the answers of the survey, it has been possible to compute the exact correction factors for about 25% of the projects in our sample. Average $\overline{\alpha}$ and $\overline{\beta}$ values have been used for the rest of the projects. The value of the $\overline{\alpha}$ factor has been calculated as the ratio between the total amount of time spent by each developer on the project and a full-time week of 40 hours. The value of the $\overline{\beta}$ factor has been calculated as the ratio between the number of respondents who declared to be actively involved in the project and the total number of respondents to our survey. Table 3 presents a summary of the overall results of the survey, along with the average values of correction factors which can be interesting benchmark values of the *SourceForge.net* repository.

Note that a possible drawback of our effort metric is that the development of a version may not start immediately after the end of the previous version. However, the branching/tagging practices that are widely adopted in OS projects imply that development activities are almost continuous [63]. The adoption of this practice has been verified through the survey mentioned above. The majority of respondents have answered that (a) there is no idle time between the development of two subsequent versions, or (b) development starts within the following 5 working days. Other answers were highly variable. Based on these results, we have decided not to correct for the idle time between versions.

*5.2. Data Analysis Tool.* Applications have been analyzed with a tool developed ad hoc. The tool provides data on the functionalities (if the application has a GUI), methods, and SLOC of application versions. The tool is capable of analyzing both the source code and the bytecode of applications written in Java.

Figure 2 shows the flow of data analyses. The tool receives the application list as input and automatically selects a *SourceForge.net* mirror. Then, it downloads the source code (or the bytecode) of all the versions of input applications into a local codebase repository. Source code (or bytecode) is analyzed and the resulting metrics are stored in the local metrics repository.

For each application version, the functionality set $\mathcal{F}$ (if a GUI is present), and the method set $\mathcal{M}$ are built by parsing the source code through a regular expression matching engine. Table 4 shows an overview of the GUI elements that are considered by the regular expression matching engine of the tool to identify functionalities, along with the related Java class. For the sake of generality, source code is analyzed by looking for class instances belonging to three tool kits (namely, AWT, Swing, and SWT).

Methods are identified by a regular expression matching engine based on a template to be used for the recognition of Java method declarations. The EBNF form of the template that we have used is

```
<modifier>* (<return_type>|void)<method_identifier>
'(' <formal_parameters_list>? ')' <throws_list>?
```

where

```
<modifier>::= public|private|protected|static|final|
native|synchronized|abstract|threadsafe|transient
<formal_parameters_list>::=
((<primitive_type>|<class_type>)
<parameter_identifier>)+
<throws_list>::= throws (<class_type>)+
```

The analysis of the Java bytecode is performed by a static analysis engine based on the Apache BCEL framework (http://jakarta.apache.org/bcel), which provides the user with a representation of the Java abstract syntax tree in a metamodel that can be used for program processing.

The source lines of code of each application have been counted using a public domain tool called SLOCCount (http://www.dwheeler.com/sloccount), which has been previously used in a number of research works and is fully conformant to the definition of physical SLOC provided by Park [59].

Data have been stored in a MySQL (http://www.mysql.org/) relational database, which has been used to compute

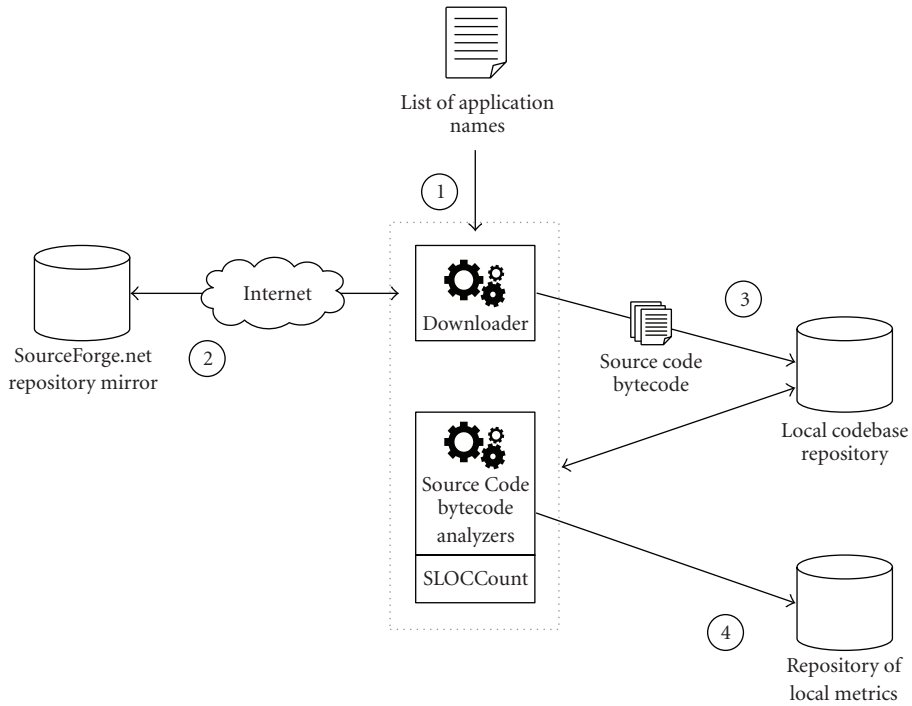


Figure 2: Action flow of the data analysis tool.

the symmetric set difference between the functionality and method sets of subsequent versions (see Section 4).

*5.3. Empirical Findings.* Hypothesis (H1) is tested by comparing the average time entropy of our sample of applications ($\overline{S}_{\text{sample}}$) with the time entropy of a reference application ($\overline{S}_{\text{model}}$) abiding by the entropy model presented in [5].

Tan and Mookerjee provide a model to simulate the evolution of a software application. This model takes into account a number of different parameters (e.g., system age, request rate for new features, learning and saturation factors, time to implement a function point, reuse factor, etc.) describing the life-cycle of an application that undergoes several maintenance initiatives. The parameters of the model are tuned based on the values provided by Tan and Mookerjee [5], consistently with previous empirical software engineering researches.

The size, the variation of methods, and the frequency of refactorings of the reference application have been set equal to the corresponding average values of our application sample. Figure 3 compares the time entropy of the reference application with the average time entropy of our sample, with $\tau$ ranging from 0 to 16. Hypothesis (H1) has been tested by means of a $z$-test with statistical significance level $\alpha = 0.01$ to check the null hypothesis $h_0$ against the alternative hypothesis $h_1$:

$$h_0 : \overline{S}_{\text{model}} \leq \overline{S}_{\text{sample}}$$
$$h_1 : \overline{S}_{\text{model}} > \overline{S}_{\text{sample}} \tag{20}$$

Given the mean values of the two data series, the $z$-test can verify whether the mean value of time entropy of the OS application sample ($\overline{S}_{\text{sample}}$) is lower than the mean value of time entropy of the reference application ($\overline{S}_{\text{model}}$). Table 5 shows the mean value and the variance of the two data series. Results show that $h_0$ must be rejected, confirming hypothesis (H1).

Hypothesis (H2) is verified by testing multiple types of correlation functions between refactoring frequency and time entropy. In order to strenghten our empirical verifications, three different relationships are tested.

(i) *Frequency of refactoring ($f_k^{\text{ref}}$) versus time entropy ($S_k$, cf. Expression* (16)*)*, in order to verify whether different frequencies of refactoring operations are related to the absolute average value of time entropy.

(ii) *Frequency of refactoring ($f_k^{\text{ref}}$) versus time entropy variation ($\Delta S_k$, cf. Expression* (18)*)*, in order to verify whether the entropy of an application is increased or decreased between subsequent versions after the execution of refactoring operations.

(iii) *Frequency of refactoring ($f_k^{\text{ref}}$) versus relative time entropy variation ($\Delta^{\%} S_k$, cf. Expression* (19)*)*, in order to verify whether the relative change of entropy between subsequent versions of an application is influenced by the occurrence of refactoring operations.

Each relationship is tested on a subset of 142 applications, selected from our sample of 393 projects by excluding applications without a GUI. Our metric of refactoring can be applied only to applications with a GUI.

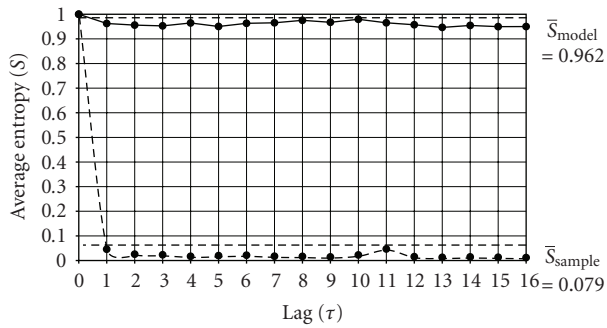


Figure 3: Time entropy of the application sample (dashed line, $\bar{S}_{\text{sample}}$) and reference application (continuous line, $\bar{S}_{\text{model}}$).

Table 4: GUI elements considered by the tool.

| | GUI Tool Kit | | |
|---|---|---|---|
| GUI Element | AWT (java.awt) | Swing (javax.swing) | SWT (org.eclipse.swt.widgets) |
| Command Button | Button | JButton | Button(SWT.PUSH) |
| Exclusive choice | Checkbox | JRadioButton | Button(SWT.RADIO) |
| Multiple choice | Checkbox | JCheckBox | Button(SWT.CHECK) |
| Drop-down list | Choice | JComboBox | Combo |
| List of items | List | JList | List |
| Single-line text entry | TextField | JTextField | Text(SWT.SINGLE) |
| Multi-line text entry | TextArea | JTextArea | Text(SWT.MULTI) |
| Password Field | TextField | JPasswordField | Text(SWT.PASSWORD) |
| Menu Item | MenuItem | JMenuItem | MenuItem |

Table 5: Statistical tests for (H1).

| Hyp | Var | Avg | $\sigma^2$ | $z_{1-\alpha}$ | $P$-value | Result |
|---|---|---|---|---|---|---|
| (H1) | $\bar{S}_{\text{model}}$ | 0.962 | $2\ 0E^{-4}$ | 2.58 | $< .001$ | Reject $h_0$ |
| | $\bar{S}_{\text{sample}}$ | 0.079 | 0 053 | | | |

Correlation functions are reported in Table 6. Data series are studied by means of regression analysis. The evaluation is performed through the analysis for each kind of relation of the coefficient of determination $R^2$, which measures the goodness of fit of a given expression to a set of data points. As shown in Table 6, none of the considered relations can be assumed to persist between refactoring frequency and time entropy ($S_k$) or the relative variation of time entropy ($\Delta^{\%}S_k$), since regressions are not statistically significant ($F$-statistic significance are greater than .05). However, the regressions between refactoring frequency and the variation of time entropy ($\Delta S_k$) are proved to be statistically significant with very low $R^2$ values (not higher than 0.23), thus confirming our hypothesis (H2). Figure 4 presents a scatter plot of the data points of refactoring frequency and the values of the corresponding dependent variables. These results support (H2), confirming that in an OS context the amount of entropy of a given system is not correlated with the frequency of refactoring operations.

In order to verify hypothesis (H3), a regression analysis has been performed on the data series of time entropy and the average unit maintenance effort variables for each application belonging to our data sample. Figure 5 provides a scatter plot of the data points of the two variables. Please note that not all applications have been found to perform refactoring operations: as a consequence, the number of points in the scatter plots are lower than 142. Results confirm that the data series can be described by means of a power relation, and exhibit a good coefficient of determination ($R^2 = 0.461$), although it cannot be considered as a *strong* correlation. The model has been proved to be statistically significant at a 99% significance level, since $P$ value $\leq .001$. These results confirm our research hypothesis (H3).

Table 7 summarizes the results of our testing for hypotheses (H1)–(H3), along with the corresponding metrics.

## 6. Discussion and Conclusions

Results indicate that community OS applications are less entropic than proprietary applications (H1). According to the definition of time entropy provided in Section 4.4, a lower level of time entropy implies a lower impact of previous maintenance operations on the effort required by subsequent maintenance operations. OS development groups are often geographically distributed and can be employed by different companies. Therefore, they cannot base coordination on face-to-face meetings [10]. Modularity and, in general, a less chaotic code is necessary to facilitate coordination without a common work environment [9]. On the contrary, traditional

Table 6: Regression analyses for (H2).

| | Dependent variable | | | | | |
|---|---|---|---|---|---|---|
| | $S_k$ | | $\Delta S_k$ | | $\Delta^{\%} S_k$ | |
| Relation | $R^2$ | Sig. | $R^2$ | Sig. | $R^2$ | Sig. |
| Linear | 0.014 | .473 | 0.160 | .012 | 0.003 | .734 |
| Logarithmic | 0.009 | .557 | — | — | — | — |
| Inverse | 0.006 | .645 | — | — | — | — |
| Quadratic | 0.014 | .774 | 0.180 | .028 | 0.007 | .884 |
| Cubic | 0.029 | .794 | 0.232 | .025 | 0.007 | .884 |
| Power | 0.003 | .739 | — | — | — | — |
| Exponential | 0.003 | .727 | 0.211 | .003 | 0.006 | .640 |
| Logistic | 0.003 | .727 | 0.211 | .003 | 0.006 | .640 |

Table 7: Summary overview of research hypotheses and results.

| Hypothesis | Metrics | Supported results |
|---|---|---|
| (H1) | Time Entropy | Community OS applications are less entropic than proprietary applications. |
| (H2) | Time Entropy<br>Refactoring Frequency | In community OS projects, time entropy cannot be related to periodic refactoring. |
| (H3) | Time Entropy<br>Unit Maintenance Effort | Higher time entropy is associated with higher unit maintenance effort. |

software developers are more likely to work in the same company and to exchange information more easily. Even though traditional software projects may be developed by groups of employees working in different locations as well, corporate structures tend to make communication more frequent and effective. On the other hand, the developers who participate to community OS projects [52, 54, 64] such as those hosted on *SourceForge.net*, generally live in different parts of the planet, with different time zones, and tend to communicate only through e-mails and forums. In addition to that, deadlines are usually tighter for traditional software projects, as they are not self-imposed, but driven by corporate schedules and revenue objectives. Consequently, software quality may be assigned a lower priority with respect to time (see also [64]).

Further, results show that refactoring operations cannot be correlated with time entropy (H2). Refactoring does not seem to be required to reduce entropy, consistent with the slower growth of entropy posited by hypothesis (H1). As noted in Section 3, refactoring can be required to integrate new contributions independently implemented by different developers. Our results seem to indicate that this may be the primary cause for refactoring in an OS context. It should also be considered that OS development practice is quite similar to agile methods [42]. Maintenance and development phase are not clearly distinct and refactoring is often used as a practice to make the application evolve, rather than to perform maintenance operations.

Results also confirm hypothesis (H3), showing how the unit maintenance costs of less entropic applications are lower than those of more entropic applications. This provides evidence supporting the cost impact of entropy. The literature on proprietary software concurs that a higher level of entropy increases the cost of subsequent maintenance operations and, thus, total development costs. Our results confirm this relationship between entropy and unit maintenance effort in the community OS development context. Given the magnitude of the effort delta, it would be interesting to verify whether development practices that keep entropy consistently lower all along the application life cycle are cost convenient. This verification is an open issue that will be considered as part of our future work (see Section 7).

Overall, results suggest that the control of the entropy level of a software system is a key issue of software development and maintenance processes. Applications with lower levels of entropy have been found to require a lower unit maintenance effort. This is of particular importance in proprietary software projects, where empirical evidence shows that entropy levels are higher compared to community OS projects. Hypothesis (H1) suggests that the adoption of development practices typical of the community OS development context can help reduce entropy. On one hand, OS developers are forced to make their code less entropic to allow different and geographically dispersed people to cooperate without meeting face to face. On the other hand, they take pride in the quality of their code especially since it is open to the judgment of other developers [8, 33, 64].

There is evidence that also closed source projects can leverage these motivational factors to reduce the entropy of software [37]. An example of how corporate interests and OS practices may merge is MySQL. MySQL is a well-known DBMS released under a dual licensing scheme, that is, both as a GPL-licensed community edition and as a commercial-licensed enterprise edition. However, 99% of the code is developed by MySQL employees. Even before its acquisition by Sun Microsystems, MySQL was managed as a traditional company and took advantage of the community as a word-of-mouth marketing channel and as a means for early and

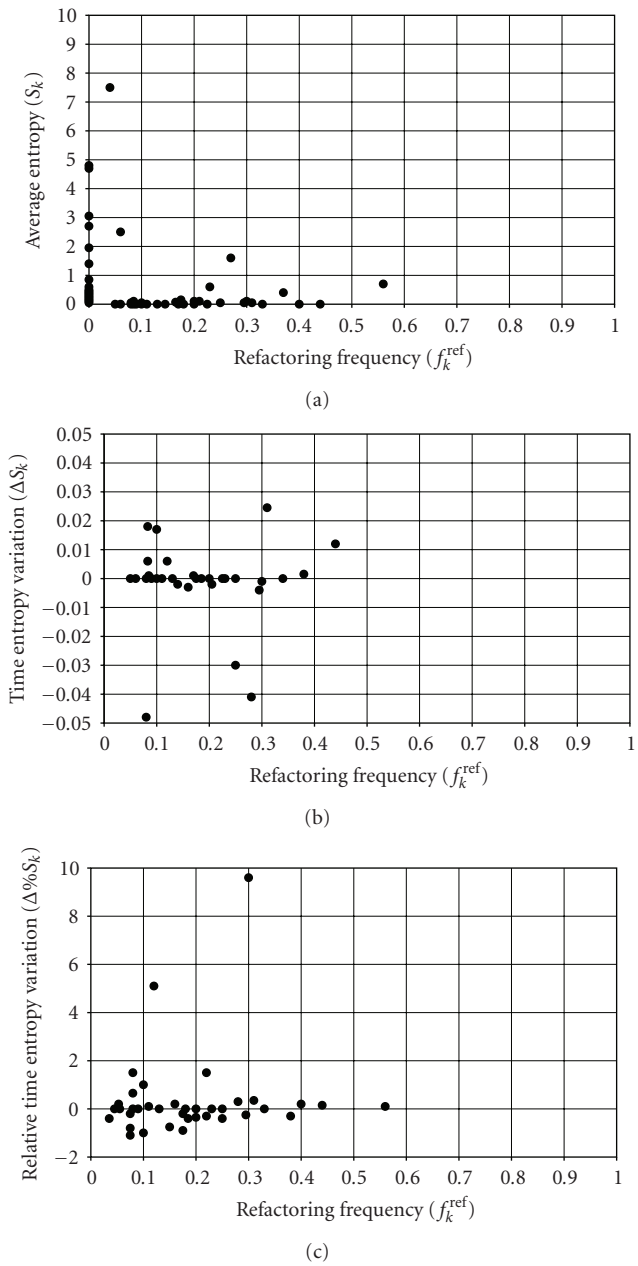


Figure 4: Scatter plots of refactoring frequency ($f_k^{ref}$) and (a) absolute average time entropy ($S_k$), (b) average time entropy variation after a refactoring operation ($\Delta S_k$), and (c) average relative time entropy variation after a refactoring operation ($\Delta^{\%} S_k$). Each data point corresponds to an application.

extensive testing. Working practices still resembled those of OS communities. Developers were located in 26 countries and worked from home. They mainly communicated through asynchronous tools, such as highly specific internal IRC channels, shared task lists, and e-mails, to overcome time zone differences. Virtual meetings over the phone or video chats were common, combined with e-mails or forum posts. This has contributed to making the quality of MySQL particularly high. For example, MySQL's code in 2005 had only 0.25 bugs per KSLOC, approximately 1/4 of the bugs of comparable applications [65].

Another significant example is SugarCRM (http://www.sugarcrm.com/), a CRM (Customer Relationship Management) engine available under both open and closed licenses, and as hosted service. Similar to MySQL, SugarCRM is a company and manages the software development process as a business. Code is controlled by employees, who are in charge of Quality Assurance, while managers formally define roadmaps and set deadlines. In addition to that, developers usually meet and work face to face in the same office. However, the external community provides an important feedback on the application, ranging from functionalities to

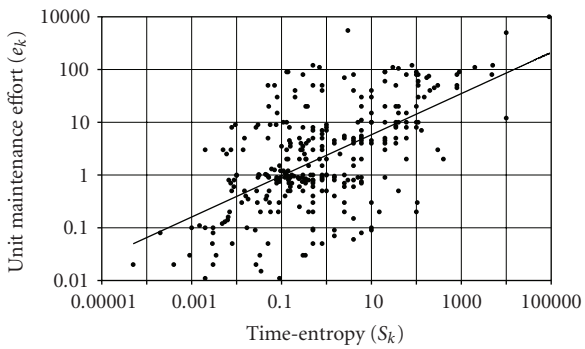


Figure 5: Scatter plot of time entropy ($S_k$) and average unit maintenance effort ($e_k$) variables. Axes in logarithmic scale, model $\overline{e}_k = 1.669 \cdot \overline{S}_k^{0.410}$, $R^2 = 0.461$, $P$ value $\leq .001$.

quality of code. Bugs, performance, functionalities, development issues, and roadmap discussions all take place in the public forum. Moreover, as 80% of the code base is open, internal developers feel exposed to the judgment of the community and are motivated to increase the quality of their code (e.g., more comprehensible and rich of comments). The adoption of OS practices has allowed SugarCRM to produce a high-quality code and, at the same time, offer a competitive service [66].

A number of other OS projects implement a mix of closed and OS practices, with similar benefits [34, 36–38, 54]. Although further research is required to verify whether the overall cost balance of lower code entropy is positive, this paper's results indicate a tangible benefit of lower entropy in terms of lower maintenance costs and associate this benefit with community OS development practices.

## 7. Threats to Validity and Future Work

We are aware that our work presents some threats to external and internal validity [67]. As regards external validity, our sample is limited to 393 *SourceForge.net* applications and may not be representative of the whole OS world. OS projects may be very different in scope, size, and governance style [52]. We have chosen to focus on *SourceForge.net* projects, which are among the most widely known and cited community-based projects, to address a significant portion of the OS world. As explained in Section 5.1, we have selected these applications by applying several significance and maturity criteria.

The verification of research hypothesis (H1) involves a comparison between community OS and proprietary applications. As we have not been able to gather data for a significant sample of proprietary applications, we have simulated the evolution of a proprietary application by referring to the model proposed by Tan and Mookerjee [5]. Although this may represent a threat to external validity, it should be noted that Tan and Mookerjee base their model on empirical analyses and previous research performed on proprietary applications and well documented in the software engineering literature (see Section 5.3 for further references).

As regards internal validity, we are aware that our results heavily rely on our measure of maintenance effort (cf. Expression (9)), which assumes that effort is proportional to the time elapsed between the release of two subsequent versions. This metric has the following drawbacks.

(i) The release of an application version may not coincide with the beginning of the development of a new version.

(ii) Open Source developers may not work full time.

(iii) Open Source developers may work for several projects in parallel.

As explained in Section 5.1, we have addressed these issues by means of a survey on OS developers. Based on this survey, our effort metric has been corrected to account for the theoretical drawbacks listed above.

We are aware that surveys are subject to a potential lack of accuracy, as both questions and answers may be misinterpreted. However, we have mitigated this threat to validity, as discussed in Section 5.1, by applying selection criteria to the recipients of the survey, by submitting only structured and quantitative questions, by manually checking all answers for consistency, and by recalling our interviewees to amend inconsistent data. Although our effort measure is still subject to some degree of approximation, it represents a direct assessment of development effort. Most cost measures in previous literature are code based and only support indirect estimates of effort [16, 68, 69]. On the contrary, as discussed in Section 2.3, a fundamental objective in the operationalization of our variables is to be independent of code.

In order to compute unit effort (cf. Expression (10)), we have considered the number of methods as a proxy of the size of an application. This represents a widely accepted proxy in previous literature [70]. Please, note that we have used the symmetric set difference rather than the arithmetic cardinality difference to evaluate the variation of size between different versions of an application. This takes into account the fact that, when developing a new version of an application, the number of new methods introduced may be equal to the number of removed methods. However, we cannot distinguish actual changes in the code of an application from the simple renaming of methods. Moreover, we have not distinguished the method count by taking into account the visibility of each application's methods, neither we have accounted for the use of external libraries. We are going to refine our empirical metrics in this way as part of our future work.

Our empirical definition of refactoring may be subject to inaccuracies as well. We discussed the limitations of our approach in Section 4.3. In particular, we plan to extend our analysis of refactoring in future work by (i) introducing a fuzzy indicator that helps to make a distinction between the maintenance effort targeted to code cleanup and actual extensions, and (ii) by evaluating the distribution of refactoring operations over time.

As discussed in Section 6, we also plan to study to what extent development practices that keep entropy consistently

lower all along the life cycle of an application have an overall positive effect on the total cost of ownership of the application.

## Appendix

This section briefly discusses the mathematical concepts and expressions used for hypotheses testing and evaluation of regression analyses.

Hypothesis (H1) is tested by performing a $z$-test to compare the mean values of two Gaussian populations with known variance. Referring to the notation of Section 5.3, the expression used to check whether the null hypothesis $h_0$ of (H1) should be rejected is

$$\frac{S_{\text{model}} - S_{\text{sample}}}{\sqrt{\sigma^2_{\text{model}}/n_{\text{model}} + \sigma^2_{\text{sample}}/n_{\text{sample}}}} \geq z_{1-\alpha}. \tag{A.1}$$

The corresponding $P$ value is computed as

$$P = 1 - \Phi\left(\frac{S_{\text{model}} - S_{\text{sample}}}{\sqrt{\sigma^2_{\text{model}}/n_{\text{model}} + \sigma^2_{\text{sample}}/n_{\text{sample}}}}\right). \tag{A.2}$$

The regression analyses used to verify hypotheses (H2) and (H3) are performed using the least squares method, and the coefficient of determination $R^2$ is computed as

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}, \tag{A.3}$$

where

$$\begin{aligned} \text{SSE} &= \sum \left(X_i - \hat{X}_i\right)^2, \\ \text{SST} &= \left(\sum X_i^2\right) - \frac{\left(\sum X_i\right)^2}{n}. \end{aligned} \tag{A.4}$$

The terms $X_i$ and $\hat{X}_i$ refer to the $i$th observed data point and the $i$th expected data point, respectively, as evaluated by means of the least squares estimator.

## Acknowledgments

The authors wish to express their thanks to Professors Carlo Ghezzi, Paolo Giacomazzi, and Laura Gotusso for their comments on the early versions of this work. They would like to thank Kaj Arno, Omer BarNir (MySQL), and Jacob Taylor (SugarCRM) for the interesting and stimulating interviews on Open Source that they had with them.

## References

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

[3] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, Englewood Cliffs, NJ, USA, 2003.

[4] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio, "Evaluating software degradation through entropy," in *Proceedings of the 7th International Software Metrics Symposium (METRICS '01)*, pp. 210–219, April 2001.

[5] Y. Tan and V. S. Mookerjee, "Comparing uniform and flexible policies for software maintenance and replacement," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 238–255, 2005.

[6] T. OReilly, "Lessons from open-source software development," *Communications of the ACM*, vol. 42, no. 4, pp. 32–37, 1999.

[7] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE Transactions on Software Engineering*, vol. 30, no. 4, pp. 246–256, 2004.

[8] E. S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, 2001.

[9] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: an empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, 2006.

[10] J. Roberts, I.-H. Hann, and S. Slaughter, "Communication networks in an open source software project," *International Federation for Information Processing*, vol. 203, pp. 297–306, 2006.

[11] V. Basili, L. Briand, S. Condon, Y. Kim, W. L. Melo, and J. D. Valett, "Understanding and predicting the process of software maintenance releases," in *Proceedings of the 18th International Conference on Software Engineering*, pp. 464–474, March 1996.

[12] Y. Zhao, H. B. Kuan Tan, and W. Zhang, "Software cost estimation through conceptual requirement," in *Proceedings of the International Conference on Quality Software*, pp. 141–144, 2003.

[13] B. Boehm, A. W. Brown, R. Madachy, and Y. Yang, "A software product line life cycle cost estimation model," in *Proceedings of the International Symposium on Empirical Software Engineering (ISESE '04)*, pp. 156–164, August 2004.

[14] C. F. Kemerer, "An empirical validation of software cost estimation models," *Communications of the ACM*, vol. 30, no. 5, pp. 416–429, 1987.

[15] L. C. Briand, K. El Emam, D. Surmann, I. Wiezczorek, and K. D. Maxwell, "An assessment and comparison of common software cost estimation modeling techniques," in *Proceedings of the International Conference on Software Engineering*, pp. 313–323, 1999.

[16] Cocomo, http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html.

[17] B. Boehm, B. Clark, E. Horowitz, R. Madachy, R. Shelby, and C. Westland, "The cocomo 2.0 software cost estimation model," International Society of Parametric Analysts, 1995.

[18] IFPUG, "Function Point Computing Practices Manual," Release 4.1. IFPUG, Westerwille, Ohio, USA, 1999.

[19] D. Garmus and D. Herron, *Function Point Analysis*, Addison Wesley, 2001.

[20] Y. Ahn, J. Suh, S. Kim, and H. Kim, "A software maintenance project effort estimation model based on function points," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 2, pp. 71–85, 2003.

[21] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Shelby, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, pp. 57–94, 1987.

[22] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley, 1981.

[23] R. Banker, S. Datar, C. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Communications of the ACM*, vol. 36, no. 11, pp. 81–94, 1993.

[24] W. Harrison, "An entropy-based measure of software complexity," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 1025–1029, 1992.

[25] S. K. Abd-El-Hafiz, "Entropies as measures of software information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '01)*, vol. 1, pp. 110–117, November 2001.

[26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 2001.

[27] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[28] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 576–585, October 2002.

[29] T. Chan, S. L. Chung, and T. H. Ho, "An economic model to estimate software rewriting and replacement times," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 580–598, 1996.

[30] J. Asundi, "The need for effort estimation models for open software projects," in *Proceedings of the Workshop Open Source Software Engineering*, pp. 1–3, 2005.

[31] J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona, "Effort estimation by characterizing developer activity," in *Proceedings of the International Workshop Economics Driven Software Engineering Research (ICSE '06)*, 2006.

[32] L. Yu, "Indirectly predicting the maintenance effort of open-source software," *Journal of Software Maintenance and Evolution*, vol. 18, no. 5, pp. 311–332, 2006.

[33] R. M. Stallman, "The GNU manifesto," http://www.gnu.org/gnu/manifesto.html.

[34] G. Goth, "Open source business models: ready for prime time," *IEEE Software*, vol. 22, no. 6, pp. 98–100, 2005.

[35] T. Wasserman, "The business of open source," in *Proceedings of the Keynote Speech International Conference on Open Source Systems*, 2006.

[36] B. Fitzgerald, "The transformation of open source software," *MIS Quarterly*, vol. 30, no. 3, pp. 587–598, 2006.

[37] R. Goldman and R. Gabriel, *Innovation Happens Elsewhere: Open Source as Business Strategy*, Morgan Kaufmann, San Francisco, Calif, USA, 2005.

[38] D. Riehle, "The economic motivation of open source software: stakeholder perspectives," *IEEE Computer*, vol. 40, no. 4, pp. 25–32, 2007.

[39] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002.

[40] R. E. Hawkins, "The economics of open source software for a competitive firm," *NETNOMICS*, vol. 6, no. 2, pp. 103–117, 2004.

[41] M. Mustonen, "When does a firm support substitute open source programming?" *Journal of Economics and Management Strategy*, vol. 14, no. 1, pp. 121–139, 2005.

[42] S. Koch, "Agile principles and open source software development: a theoretical and empirical discussion," in *Extreme Programming and Agile Processes in Software Engineering*, pp. 85–93, 2004.

[43] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison Wesley, 2002.

[44] D. Kafura and G. R. Reddy, "The use of software complexity metrics in software maintenance," *IEEE Transactions on Software Engineering*, vol. 13, no. 3, pp. 335–343, 1987.

[45] H. D. Rombach, "A controlled experiment on the impact of software structure on maintainability," *IEEE Transactions on Software Engineering*, vol. 13, no. 3, pp. 344–354, 1987.

[46] H. Zuse, *Software Complexity: Measures and Methods*, Walter de Gruyter, Hawthorne, NJ, USA, 1991.

[47] N. E. Benton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[48] A. E. Hassan and R. Holt, "The chaos of software development," in *Proceedings of the International Workshop Principles Software Evolution*, pp. 84–95, 2003.

[49] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, Urbana, Ill, USA, 1949.

[50] Free Software Foundation, "Philosophy of GNU project," http://www.gnu.org/philosophy/.

[51] K. Crowston and J. Howison, "The social structure of free and open source software development," *First Monday*, vol. 10, no. 2, 2005.

[52] E. Capra and A. I. Wasserman, "A framework for evaluating managerial styles in open source projects," *IFIP International Federation for Information Processing*, vol. 275, pp. 1–14, 2008.

[53] K. Fogel, "Producing Open Source software," OReilly, Sebastopol (CA), 2006.

[54] A. I. Wasserman and E. Capra, "Evaluating software engineering processes in commercial and community open source projects," in *Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS '07)*, May 2007.

[55] P. Gomes and N. Joglekar, "The costs of coordinating distributed software development tasks," Tech. Rep., Boston University School of Management, 2004.

[56] I. Tervonen and P. Kerola, "Towards deeper co-understanding of software quality," *Information and Software Technology*, vol. 39, no. 14-15, pp. 995–1003, 1998.

[57] J. A. Roberts, I.-H. Hann, and S. A. Slaughter, "Understanding the motivations, participation, and performance of open source software developers: a longitudinal study of the Apache projects," *Management Science*, vol. 52, no. 7, pp. 984–999, 2006.

[58] K. Beck, *Extreme Programming Explained*, Addison Wesley, 1999.

[59] R. E. Park, "Software size measurement: a framework for counting source code statements," Tech. Rep. CMU/SEI-92-TR-020, Carnegie Mellon University, 1992.

[60] A. Papoulis and S. U. Pillai, *Probability, Random Variables and Stochastic Processes*, McGraw-Hill, 2001.

[61] W. Humphrey, *Managing the Software Process*, Addison-Wesley, 1990.

[62] J. Howison and K. Crowston, "The perils and pitfalls of mining source forge," in *Proceedings of the International Workshop Mining Software Repositories*, pp. 7–12, 2004.

[63] E. S. Raymond, *The Art of Unix Programming*, Addison-Wesley Professional, 2003.

[64] E. Capra, *Analysis of the impact of different OS managerial approach on software design quality and effort*, Ph.D. thesis, Politecnico di Milano, Department of Electronics and Information, 2008.

[65] Coverity, "Analysis of MySQL," February 2005, http://www.coverity.com/.

[66] M. Goulde and J. Mulligan, "SugarCRM finds the sweet spot in customer relationship management," Forrester Research Case Study.

[67] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*, The Kluwer International Series in Software Engineering, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[68] A. J. Albrecht, "Measuring applications development productivity," in *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, 1979.

[69] A. J. Albrecht and J. E. Gaffney Jr., "Software function, source lines of code and development effort prediction: a software science validation," *IEEE Transactions on Software Engineering*, vol. 9, no. 6, pp. 639–648, 1983.

[70] P. M. Johnson, C. A. Moore, J. A. Dane, and R. S. Brewer, "Empirically guided software effort guesstimation," *IEEE Software*, vol. 17, no. 6, pp. 51–56, 2000.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 517532, 12 pages
doi:10.1155/2010/517532

*Research Article*

# Improvement of Open Source Software Usability: An Empirical Evaluation from Developers' Perspective

**Arif Raza,[1] Luiz F. Capretz,[1] and Faheem Ahmed[2]**

[1] *Department of Electrical & Computer Engineering, University of Western Ontario, London, ON, Canada N6A 5B9*
[2] *Faculty of Information Technology, United Arab Emirates University, P.O. Box 17551, Al Ain, UAE*

Correspondence should be addressed to Arif Raza, araza7@uwo.ca

Received 30 January 2010; Accepted 7 July 2010

Academic Editor: Hongyu Zhang



User satisfaction has always been important for software success whether it is Open Source Software (OSS) or closed proprietary software. Even though we do not presume that OSS always has poor usability, as there are examples of good usable open source software, it would still be agreed that OSS usability has room for further improvement. This paper presents an empirical investigation to study the impact of some key factors on OSS usability from developers' points of view. This is one of the series of four studies that we are conducting regarding improvement of OSS usability from OSS developers, users, contributors, and industry perspectives. The research model of this empirical investigation studies and establishes the relationship between the key usability factors from developers' perspective and OSS usability. A data set of 106 OSS developers from 18 open source projects of varied size has been used to study the research model. The results of this study provide empirical evidence that the studied key factors play a significant role in improving OSS usability.

## 1. Introduction

The term open source software refers to software equipped with licenses that provide existing and future users the right to use, inspect, modify, and distribute (modified and unmodified) versions of the software to others. It is not only the concept of providing "free" access to the software and its source code that makes OSS the phenomenon that it is, but also the development culture [1]. Open source is a software development method that makes source code available to a large community that participates in development by following flexible processes and communicating via the Internet [2]. The favorable acceptance of OSS products by business and the direct involvement of major IT vendors in OSS development have transformed OSS from a fringe activity, developed for public good, to a mainstream, commercially viable form [3]. The collaborative nature of the OSS culture makes use of a wide volunteer community, which conducts its development activities in a decentralized environment that has the direct result of effectively lowering production costs and improving the software quality [4].

The International Organization for Standardization and The International Electrotechnical Commission ISO/IEC 9126-1 [5] categorize software quality attributes into six categories: functionality, reliability, usability, efficiency, maintainability, and portability. In the standard, usability is defined as "*The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.*" However usability is probably the least addressed area in OSS research and development. In its early days it was generally believed that OSS is for the technically adept users; that resulted in a blurred boundary between developers and users. Thus, usability has never been on the top priority list of OSS developers. Nichols and Twidale [6] have come up with another reason why usability aspects are not so enthusiastically addressed as compared to functionality issues. According to them it is because the latter have more challenges and recognition factors from the community in them. Being voluntary in nature of work, in OSS projects usability problems are found less interesting and less challenging.

Hedberg et al. [7] point out that it is no more the case as in the past when OSS users were the codevelopers who

used to expect frequent crashing of the applications and bugs in the code. They feel that although so far usability in OSS has not been tested enough, now OSS community has started to realize that their target audiences are no longer their codevelopers only. As a result, OSS systems need to be designed, keeping in mind the requirements, expectations, and demands of a common nontechnical user. Viorres et al. [8] also highlight a general trend in OSS development that is instead of following software engineering (SE) practice of design, specifications, testing, and prototyping; most OSS systems follow a "*bottom-up*" approach where the focus is on the development on technical issues and individual components whereas the modeling of the whole system comes later; plus, user interface and related issues get a relatively lower priority.

This paper contributes in increasing the understanding of the effects of some key usability factors through empirical investigation that they play a vital role in improving OSS usability. A quantitative survey of developers involved in different OSS projects has been conducted and reported here. The survey has been used to analyze the conceptual model and hypotheses of the study. The results provide the evidence that the stated key factors play an important role towards the improvement of OSS usability.

In Section 2, we present the literature review that motivated this research work as well as helped in selecting the key factors for the study. Section 3 illustrates the research model and the hypotheses of this study. Section 4 explains the research methodology, data collection process, and the experimental setup in its first part, reliability and validity analysis of the measuring instrument in the second, and data analysis procedures in its third part. In Section 5, we present the hypotheses testing and the analysis of the results. It is followed by the discussion in Section 6 that also includes the limitations of the study. Finally the paper concludes in Section 7.

## 2. Literature Review

*2.1. Research Motivation and Related Work.* Empirical studies regarding open source quality assurance activities and quality claims are rare [9]. Koponen [10] discusses defect management and version management system as an integral part of OSS maintenance process. Aberdour [11] observes that the open source software model has led to the creation of significant pieces of software, and many of these applications show levels of quality comparable to closed source software development. Raymond [4] suggests that the high quality of OSS can be achieved due to high degree of peer review and user involvement in bug/defect detection. Generally a popular or active project means that the community in the OSS project are interacting constantly and providing feedback to activities such as defect identification, bug fixing, new feature request, and support requests for the further improvement.

Wayner [12] finds that developers contribute from around the world, meet face-to-face infrequently if at all, and coordinate their activity primarily by means of computer-mediated communications. Crowston and Scozzi [13] investigate the coordination practices for software bug fixing in OSS development teams and observe that task sequences are mostly sequential and composed of a few steps, namely, submit, fix, and close, and effort is not equally distributed among process actors; as a result, a few contribute heavily to all tasks while the majority just submit one or two bugs. Cubranic and Booth [14] discuss major issues of coordinating open source development projects, including collaborative communication mediums and configuration management tools. Mockus et al. [15] provide a comprehensive comparison of Apache against five commercial products in terms of developer participation, team size, productivity and defect density, and problem resolution. The Floss Survey [16] identifies many other reasons why developers are involved in OSS development, including becoming part of the open source community, promoting the open source mode of development, supporting the idea of "*free*" as an alternative to proprietary software, gaining a reputation, and having fun. In proprietary software, software quality testing is limited within a controlled environment and specific scenarios [17]. However, OSS development involves much more elaborate testing as OSS solutions are tested in various environments, by various skills and experiences of different programmers, and are tested in various geographic locations around the world [17–20]. According to Feller and Fitzgerald [21], OSS is characterized by active developers' community living in a global virtual boundary. OSS has emerged to address common problems of traditional software development that includes software exceeding its budget both in terms of time, and money, plus making the production of quick, inexpensive, and high quality reliable software possible. Earlier, OSS was more about operating systems and development tools. However lately, entertainment applications have also been developed. Independent peer review by codevelopers in OSS makes its quality presumably better and is also proved by their achievement of "*significant market share without any conventional marketing or advertising campaigns*."

Koppelman and Van Dijk [22] focus on the role of clients and users in projects, how to deal with different stakeholders who look at the product from a different perspective, how to communicate with them, and how to involve the real users and clients in the design process. They suggest that the designers should not simply rely on their own experiences and instincts. Golden et al. [23] support the idea of addressing usability issues at software architecture design level. They believe that separating usability concerns from functionality at architecture level in order to consider them at a later stage of testing does not work. Rather, this approach leads to extensive restructuring and even "*re-architecting of software systems*." They have come up with Usability-Supporting Architectural Pattern (USAP) that supports specific usability issues at architecture level. Although they themselves state that USAP is quite detailed and complex to imply as a whole, they conclude basing on a test case study that it is a beneficial tool to address specific usability issues in software architecture designs. They also observe that usability concerns could be better addressed if "*implications of usability heuristics for software design*" are made clear and explicit to the software designers.

Nichols and Twidale [24] feel that usability can hardly be considered a resolved issue in proprietary software environment that has better resources, let alone in OSS that has relatively less resources and where most work is done on voluntary basis with no monetary benefits or rewards. Another factor they identify is the lack of resources in OSS to achieve high quality, particularly in the context of usability, as compared to the closed proprietary software. To have more participation to analyze and fix usability bugs, what is required is to make usability reporting easier plus have less efforts and lower "*cultural, technical and usability barriers*." Unlike functionality bugs, where duplication in bug reporting does not help, a large number of usability bug reporting can help in prioritizing the usability-related errors, to be fixed. However it is required to have some way to speed up the discussion about usability-related issues and to have an easier and faster solution and consensus.

Çetin and Göktürk [25] observe that being a non-functional quality attribute, usability cannot be measured directly; it could be measured through users' feedback and cognitive walkthroughs. So far, there seems to be no metrics available for the OSS developers against which they could measure usability of their projects. A standardized user interface guideline may be developed by usability experts that the developers can adhere to, in order to have consistency and conformity in the designs.

Zhao and Deek [26] hypothesize that exploratory method is an effective way to impart such knowledge to the users so that they could be able to inspect and report usability errors in OSS and hence play their part in a better way, towards OSS usability improvement. A model has also been proposed by them to adapt the exploratory learning method for such purposes.

Hussain et al. [27], in their recent survey about the integration of agile methods and usability, conclude that the integration of agile methods with usability/user-centered design not only adds value to the adopted processes and to the teams of the respondents but also increases the satisfaction of the end-users of the product developed.

Hedberg et al. [7] observe that there is a lack of strict plan and design process in OSS environment as software development mainly relies on developers' skills. Advocating the early user feedback, the authors recommend to "*understand and specify the user, his/her work practice/tasks and the context of use, and carefully redesign the work practice/tasks based on the understanding, actively involve the user, gather early user feedback and iterate the design solution based on the user feedback*."

The above literature review and recommendations have played a motivating role in this study. We have been able to identify some usability factors and analyze and validate them empirically based on OSS developers' perception, as presented in the following sections.

*2.2. Usability Factors: Literature Review of Concepts.* Understanding *users' requirements* and expectations by OSS developers is an important issue that needs to be addressed seriously. Realizing the different intuitive approach of programmers from that of end-users, Pemberton [28] observes that while developing software they are normally satisfied with its usability and interface. Referring to the problems in an OSS environment, he states: "*the general public will have an itch they cannot scratch; the programmers will not have that itch, and so will not scratch it*." Nichols and Twidale [24] also identify that generally developers do not realize the needs and expectations of end-users that may lead to poor usability in OSS. They refer to Human Interface Guidelines (HIGs) that cannot only prevent such discussions but can also be considered as an authority on what will be done. Çetin and Göktürk [25] also realize that the main theme of OSS is the software development through collaboration and cooperation. Traditionally, OSS users have had technical and computer-oriented background and needed less effort to use OSS systems like Apache, GNU C compilers, bash shell, and so forth. However, as OSS has become more popular, more need is being felt to have usable systems. Koppelman and Van Dijk [22] stress that software developers should not simply rely on their own experiences and instincts. They should learn how to communicate with users in order to better understand their expectations.

The importance of HCI and *Usability Experts' Opinion* cannot be undermined. Big commercial organizations generally employ such experts to address usability issues in their projects. However their representation is generally missing in OSS projects probably due to voluntary work environment of OSS. Nichols and Twidale [6] identify why usability experts are not generally involved in OSS projects, mainly because there are fewer such experts in OSS world; they are not "*incentivised by the OSS approach in the way that many hackers are,*" and they do not find themselves "*welcomed into OSS projects*." Hedberg et al. [7] emphasize on the need of usability experts' contribution and show concern regarding their lack of participation in OSS development. They point out that OSS users may report usability-related problems and bugs but without formal training, neither the users nor the developers can fix them. An expert's opinion and suggestion is thus required; that is currently missing from the scene of OSS development. They propose the incorporation of usability guidelines and active participation of usability experts in OSS projects, possibly from the platforms of large commercial organizations, as they have also started participating in OSS development.

*Incremental design approach*, that is, introduction of advance features of software to users in an incremental way would make them more comfortable. Gaming softwares use this approach all the time and allow their users to face advance levels in an incremental and gradual fashion. OSS developers need to realize this fact, as well, that their target audiences may include novice users for whom the software application would be more adaptable if advance features are introduced in a gradual and progressive way. Yunwen and Kishida [29] highlight the need of modularized software system design to enable the end-users to encounter the difficulty levels gradually and progressively. They believe that modularized OSS system architecture design with progressive introduction of difficult and advance features would attract more users. Aberdour [11] also finds code modularity a convenient way to add new features in software.

It reduces code complexity and allows different programmers to extend the program by working in parallel and without interfering in others' work.

Usability aspects cannot be improved in OSS unless there are ways to test and measure them quantitatively. Çetin and Göktürk [25] highlight the importance of testing and measurement by stating: "*one cannot improve what is not measured*." Holzinger et al. [30] observe that "*the evaluation of consistency within an e-learning system and the ensuing eradication of irritating discrepancies in the user interface redesign is a big issue*." They have also come up with the Shadow Expert Technique (SET) to evaluate the consistency of the user interface and have applied it to a university learning management system. Nichols and Twidale [6] identify that fixing an interface needs an extra care so that it should not lead to inconsistency as "*a major success criterion for usability is consistency of design*." Usability problems are neither easier to specify nor very convenient to be fixed, particularly considering virtual boundaries of OSS where developers mostly do their work autonomously. In their other study, Nichols and Twidale [24] observe the bias in treating usability bugs as compared to functionality bugs that could crash the system. Usability issues, as expected, are more subjective in nature and more debatable as a user interface element may be more confusing to some people and less to others. Such issues could prolong the discussion of analyzing and fixing usability bugs. To have more participation to analyze and fix usability bugs, what is required is to make usability reporting easier plus have less efforts and lower "*cultural, technical and usability barriers*." Unlike functionality bugs, where duplication in bug reporting does not help, a large number of usability bug reportings can help in prioritizing the usability bugs to be fixed. Hedberg et al. [7] suggest evaluation methods under the guidance of usability experts, usability testing, and bug reporting. They feel the need of in-depth empirical research to understand the challenges related to usability and quality assurance in OSS. Viorres et al. [8] also highlight a few OSS usability issues such as to improve bug reporting facilities in software, to improve the analysis procedure of usability errors by OSS community through application of human computer interaction (HCI) principles, and to support argumentation to resolve such issues.

As a long-term solution, students of the Software Engineering and Computer Science disciplines should be taught how to address user-centric issues in their software development projects to increase their understanding of the users' point of view. They should be encouraged to appreciate the fact that finding a solution to a particular programming problem is not the ultimate goal. They should rather come up with design that could meet the expectations of end-users. Faulkner and Culwin [31] observe that HCI and Software Engineering educators have always been in different camps. Although the growth of HCI in terms of books and as a subject taught in computer science courses is the recognition of importance of HCI, they suggest that there is a need of more interaction between HCI and SE by adopting HCI as the underlying principle to the systems development. According to them, the aim of usability engineering education must be to ensure that effectiveness, efficiency, and user satisfaction are present in software. The guidance from HCI specialist needs be provided to the software developer in a useful form, which is only possible through the unification of knowledge and vocabulary of both. However, Rosson et al. [32] realize that the main challenge in teaching usability engineering is to come up with realistic projects for the students, such that meaningful issues could be addressed in a manageable time of a semester.

Markov [33] argues that usability is about "*total user experience*," not only about the user interface, as it is commonly but incorrectly assumed. It should be involved in all the phases of the product such as installation, use, and maintenance. Although it is not the case that every OSS must have a poor user interface, usability of OSS projects requires improvement, in general. In their research work, Nichols and Twidale [6] observe that OSS is growing and has developed a repute of being reliable, efficient, and functional. However, still common novice computer user prefers to use proprietary software for many reasons: their better usability is one of them. They talk about usability from applications like word processors and web mail servers which are basically aimed at serving a common user. They also realize that, considering fewer resources of OSS, it could take long for an OSS project to be mature and comparable with closed proprietary software. Another point they make is that, in OSS culture, coding starts earlier and refinement of design depends on constant reviews. They advocate that to improve OSS usability, designing of interface should be done before the start of the coding, to keep it consistent. Viorres et al. [8] refer to various reasons why software developers go for OSS. These include educational reasons, reusability, and developing reputation. However, they highlight concerns about software usability and complexity in installation and maintenance of OSS development tools, their nonadherence to backwards compatibility, and limited documentation. Hedberg et al. [7] propose the adaptation of proven methods in OSS environment to ensure higher quality and address usability issues. Holzinger et al. [34] discuss a user-centered system developed at the clinical department of dermatology at the Medical University Hospital in Graz. The system not only improved the existing system but also helped elderly people to overcome their computer fear.

## 3. Research Model and the Hypotheses

In this paper, we present a research model to analyze the relationship between the key usability factors and the open source software usability. This work empirically investigates the association between these key usability factors and the OSS Usability. The theoretical model to be empirically tested in this paper is shown in Figure 1. Our aim is to investigate the answer to the following research question:

*How OSS developers can improve software usability?*

There are five independent and one dependent variables in this research model. The five independent variables are called

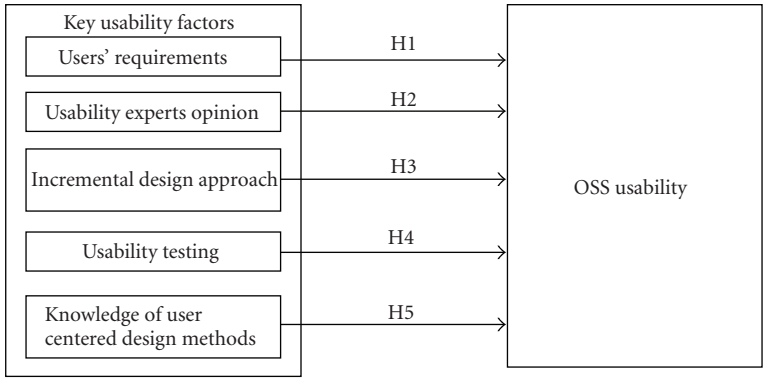


Figure 1: Research Model.

*"key usability factors"* in the rest of this paper. They include users' requirements, usability experts' opinion, incremental design approach, usability testing, and knowledge of user-centered design methods. The dependent variable of this study is the OSS usability. The multiple linear regression equation of the model is as follows:

$$\text{OSS Usability} = f_0 + f_1 v_1 + f_2 v_2 + f_3 v_3 + f_4 v_4 + f_5 v_5, \quad (1)$$

where $f_0$, $f_1$, $f_2$, $f_3$, $f_4$, and $f_s$ are the coefficients and $v_1$, $v_2$, $v_3$, $v_4$, and $v_s$ are the five independent variables. In order to empirically investigate the research question we hypothesize the following.

(H1) understanding users' requirements by the software developers is positively related with improving usability in OSS.

(H2) seeking usability experts' opinion by the software developers is positively related with improving usability in OSS.

(H3) incremental approach in OSS design plays a positive role in improving usability in OSS.

(H4) usability testing by project managers/software developers has a positive impact on usability in OSS.

(H5) knowledge of user-centered design (UCD) methods is positively related with improving software.

## 4. Research Methodology

Open source software projects deal with different categories of applications like communications, database, desktop environment, education, financial, games/entertainment, networking, and so on. We sent personalized emails to OSS developers of different projects on sourceforge.net. The projects differed in size and range from small to large scale. However, we selected the projects having activity of 90% and more. We sent our questionnaire to the OSS developers working on the projects in the categories of Database (118), desktop environment (127), development (135), testing (83), communications (104), games/entertainment (309), education (309), financial (236), and enterprise (35) as shown in Figure 2.

We assured the participants that our survey neither required their identity nor would it be recorded. However to support our analysis of data in terms of experience of the developers and the project size, they have been working on, we asked them to share with us their OSS development experience and their development team size. These two questions were optional for the participants to respond to unlike the questions related to OSS usability which were mandatory to respond to in the survey. We received 106 responses altogether and 104 of them chose to respond to these two questions. 63 of the 104 respondents had less than or equal to 5 years of OSS development experience; 31 had more than 5 years but less than or equal to 10 years of experience whereas 10 of the respondents stated that they had more than 10 years of experience in OSS development as reflected in Figure 3.

In the survey, 56 respondents had less than or equal to 10 team members as developers in their project, 27 had more than 10 but less than or equal to 20 team members as developers, and 21 had more than 20 members in their development team as represented graphically in Figure 4.

The above statistics have been presented to reflect the experience of the respondents as well as the size of the OSS project they belong to.

*4.1. Data Collection and the Measuring Instrument.* In this study, we have collected data on the key usability factors and the perceived level of OSS usability improvement. The questionnaire presented in the appendix requires respondents to indicate the extent of their agreement or disagreement with statements using a five-point Likert scale. The Likert scale ranges from "strongly agree" (1) to "strongly dis-agree" (5) for all items associated with each variable. For each of the independent variables as well as the dependent variable, four statements are presented. These statements elaborate the specific key factor and its related issues. The statements are designed to collect measures on the extent to which the variable is practiced within each project. We have thus used twenty separate items to measure the independent variables and four items to measure OSS developers' point of view regarding OSS usability improvement. Although not much work on such lines is available, we have reviewed

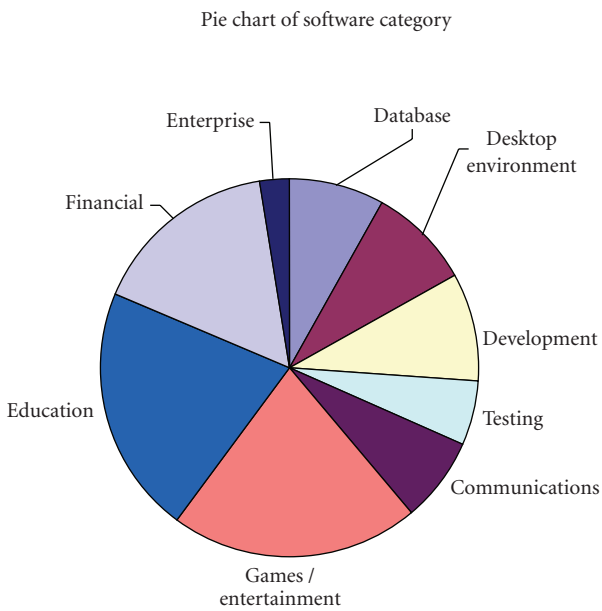


Figure 2

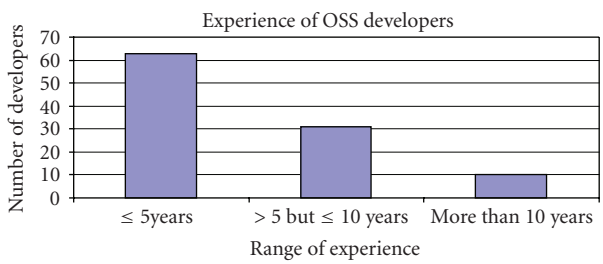


Figure 3

previous researches on the subject of OSS usability, so that a comprehensive list of measuring factors could be constructed.

*4.2. Reliability and Validity Analysis of Measuring Instrument.* The two integral features of any empirical study are reliability, which refers to the consistency of the measurement, and the validity, that is the strength of the inference between the true value and the value of a measurement. For this empirical investigation, we have used the most commonly used approaches in empirical studies to conduct reliability and validity analysis of the measuring instruments. The reliability of the multiple-item measurement scales of the five usability factors is evaluated by using internal-consistency analysis, which is performed using coefficient alpha [35]. In our analysis, the coefficient alpha ranges from 0.55 to 0.67 as shown in Table 1. van de Ven and Ferry [36] state that a reliability coefficient of 0.55 or higher is satisfactory, and Osterhof [37] suggests that 0.60 or higher is satisfactory. Therefore, we conclude that the variable items developed for this empirical investigation are reliable.

Campbell and Fiske [38] state that convergent validity occurs when the scale items are correlated and move in the same direction in a given assembly. The principal component analysis [39] is performed for all five key usability factors and reported in Table 1. We have used eigenvalue [40] as a reference point to observe the construct validity using principal component analysis. In this paper, we have used eigenvalue-one criterion, also known as Kaiser Criterion [41, 42], which means any component having an eigenvalue greater than one is to be retained. Eigenvalue analysis reveals that four out of five variables completely form a single factor whereas eigenvalue for the usability testing is 0.99, that is very close to the threshold of 1.0. Therefore, the convergent validity has been regarded as sufficient.

Table 1: Coefficient alpha and principal component analysis (PCA) of variables.

| Usability factors | Item no. | Coefficient $\alpha$ | PCA eigen value |
|---|---|---|---|
| Users requirements | 1–4 | 0.67 | 2.19 |
| Usability experts opinion | 5–8 | 0.64 | 1.22 |
| Incremental design approach | 9–12 | 0.55 | 1.08 |
| Usability testing | 13–16 | 0.55 | 0.99 |
| Knowledge of UCD methods | 17–20 | 0.59 | 1.05 |

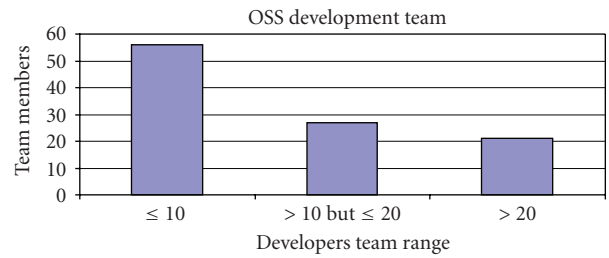


Figure 4

*4.3. Data Analysis Procedure.* We have analyzed the research model and the significance of hypotheses H1–H5 through different statistical techniques in three phases. In phase I, we have used normal distribution tests and parametric statistics whereas, in phase II, nonparametric statistics have been implemented. Due to a relatively small sample size, both parametric as well as nonparametric statistical approaches are used to reduce the threats to external validity. As our measuring instrument has multiple items for all the five independent variables as well as the dependent variable (refer to the appendix), their ratings by the respondents are summed up to get a composite value for each of them. Tests are conducted for hypotheses H1–H5 using parametric statistics by determining the Pearson correlation coefficient. For nonparametric statistics, tests are conducted for hypotheses H1–H5 by determining the Spearman correlation coefficient. To deal with the limitations of a relatively small sample size and to increase the reliability of the results, hypotheses H1–H5 of the research model are tested using partial least square (PLS) technique in phase III. According to Fornell and Bookstein [43] and Joreskog

and Wold [44], the PLS technique is helpful in dealing with issues such as complexity, nonnormal distribution, low theoretical information, and small sample size. The statistical calculations are performed using minitab-15. ( *Minitab is a statistics software package (see* http://en.wikipedia.org/wiki/List_of_statistical_packages*) and is often used in conjunction with the implementation of Six Sigma (see* http://en.wikipedia.org/wiki/Six_Sigma*), CMMI (see* http://en.wikipedia.org/wiki/CMMI*), and other statistics-based process improvement methods. Minitab is available in 7 different languages.* )

## 5. Hypotheses Testing and Results

*5.1. Phase I.* To test hypotheses H1–H5 of the research model (shown above in Figure 1), parametric statistics is used in this phase by examining the Pearson correlation coefficient between individual independent variables (key usability factors) and the dependent variable (OSS usability). The results of the statistical calculations for the Pearson correlation coefficients are displayed in Table 2. It is to be noted that, "*In statistical* (*see* http://en.wikipedia.org/wiki/Statistics) *hypothesis testing* (*see* http://en.wikipedia.org/wiki/Hypothesis_test), *the P-value is the probability* (*see* http://en.wikipedia.org/wiki/Probability) *of obtaining a test statistic.* (*see* http://en.wikipedia.org/wiki/Test_statistic) *The lower the P-value, the less likely the result is if the null hypothesis is true, and consequently the more "significant" the result is, in the sense of statistical significance* (*see* http://en.wikipedia.org/wiki/Statistical_significance) " [45].

The Pearson correlation coefficient between users' requirements and OSS usability is found positive (0.264) at $P < .05$ and hence justified hypothesis H1. The Pearson correlation coefficient of 0.084 is observed at $P = .393$ between usability experts' opinion and OSS usability and hence found insignificant at $P < .05$. Therefore hypothesis H2 that deals with usability experts' opinion and OSS usability is rejected. Hypothesis H3 is accepted based on the Pearson correlation coefficient (0.274) at $P < .05$, between the incremental design approach and OSS usability. The positive correlation coefficient of 0.338 at $P < .05$ is also observed between the OSS usability and usability testing which meant that H4 is accepted. Hypothesis H5 is found significant too and thus accepted after analyzing the Pearson correlation coefficient of 0.439 at $P < .05$ between knowledge of UCD methods and OSS usability. Hence, as observed and reported above, hypotheses H1, H3, H4, and H5 are found statistically significant and are accepted whereas H2 is not supported and is therefore rejected.

*5.2. Phase II.* Nonparametric statistical testing is conducted in this phase by examining Spearman correlation coefficient between individual independent variables (key usability factors) and the dependent variable (OSS usability). The results of the statistical calculations for the Spearman correlation coefficient are also displayed in Table 2.

The Spearman correlation coefficient between users' requirements and OSS usability is found positive (0.480) at $P < .05$ and hence justified hypothesis H1. For hypothesis H2, the Spearman correlation coefficient of 0.122 is observed with $P = .213$; hence at $P < .05$ no significant relationship is found between usability experts' opinion and OSS usability in this test as well. Hypothesis H3 is accepted based on the Spearman correlation coefficient (0.420) at $P < .05$, between the incremental design approach and OSS usability. The positive Spearman correlation coefficient of 0.390 at $P < .05$ is also observed between the OSS usability and usability testing, which means that H4 is accepted. Hypothesis H5 is found significant too and thus accepted after analyzing the Spearman correlation coefficient of 0.485 at $P < .05$ between knowledge of UCD methods and OSS usability.

Table 2: Hypotheses testing using parametric and nonparametric correlation coefficients.

| Hypothesis | Usability factor | Pearson correlation coefficient | Spearman correlation coefficient |
|---|---|---|---|
| H1 | Users requirements | 0.264* | 0.480* |
| H2 | Usability experts opinion | 0.084** | 0.122** |
| H3 | Incremental design approach | 0.274* | 0.420* |
| H4 | Usability testing | 0.338* | 0.390* |
| H5 | Knowledge of UCD methods | 0.439* | 0.485* |

* Significant at $P < .05$. ** Insignificant at $P > .05$.

Hence, as observed and presented above, H1, H3, H4, and H5 are found statistically significant and are accepted whereas H2 is not supported and hence rejected in nonparametric analysis, as well.

*5.3. Phase III.* In order to do the cross-validation of the results obtained in Phase I and Phase II, partial least square (PLS) technique has been used in this phase of hypotheses testing. The direction and significance of hypotheses H1–H5 are examined. In PLS, the dependent variable of our research model, that is, OSS usability, is placed as the response variable and independent key usability factors as the predicate. The test results that contain observed values of path coefficient, $R^2$, and $F$-ratio are shown in Table 3. The "*users' requirements*" is observed to be significant at $P < .05$ with path coefficient of 0.302, $R^2$ of 0.070, and $F$-ratio of 7.782. Usability experts' opinion has path coefficient of 0.129 with $R^2$ of 0.007, and F-ratio of 0.737 and is found insignificant at $P < .05$ (with observed $P = .393$). Incremental design approach is observed to have the same direction as proposed in hypothesis H3 with path coefficient 0.244, $R^2$ 0.075, and $F$-ratio 8.429 at $P < .05$. Usability testing is also found in conformance with hypothesis H4 with observed values of path coefficient of 0.310, $R^2$ of 0.114, and $F$-ratio of 13.412 at $P < .05$. And finally knowledge of UCD methods (path coefficient: 0.446, $R^2$: 0.193, and $F$-ratio: 24.888 at $P < .05$) is also found in accordance with H5. Hence in this phase, like in phase I and phase II, hypothesis H2 that deals with

Table 3: Hypotheses testing using PLS regression.

| Hypothesis | Usability factor | Path coefficient | $R^2$ | $F$-ratio |
|---|---|---|---|---|
| H1 | Users' requirements | 0.302 | 0.070 | 7.782* |
| H2 | Usability experts opinion | 0.129 | 0.007 | 0.737** |
| H3 | Incremental design approach | 0.244 | 0.075 | 8.429* |
| H4 | Usability testing | 0.310 | 0.114 | 13.412* |
| H5 | Knowledge of UCD methods | 0.446 | 0.193 | 24.888* |

* Significant at $P < .05$. ** Insignificant at $P > .05$.

Table 4: Multiple linear regression analysis of the research model.

| Model coefficient name | Model coefficient | Coefficient value | $t$-value |
|---|---|---|---|
| Users' requirements | $f_1$ | 0.277 | 2.800* |
| Usability experts opinion | $f_2$ | −0.006 | −0.045** |
| Incremental design approach | $f_3$ | 0.116 | 1.218* |
| Usability testing | $f_4$ | 0.111 | 1.097* |
| Knowledge of UCD methods | $f_5$ | 0.355 | 3.740* |
| Constant | $f_0$ | 1.796 | 1.003* |

* Significant at $P < .05$. ** Insignificant at $P > .05$.

usability experts' opinion and OSS usability is not found to be statistically significant at $P < .05$.

*5.4. Testing of the Research Model.* The multiple linear regression equation of our research model is depicted by (1). The purpose of research model testing is to provide empirical evidence that our key factors play a significant role in improving open source software usability. The testing process consists of conducting regression analysis and reporting the values of the model coefficients and their direction of association. OSS usability is placed as response variable and key factors as predicators. Table 4 displays the regression analysis results of the research model. The path coefficient of four out of five variables, users' requirements, incremental design approach, usability testing, and knowledge of user-centered design methods, is found positive, and their $t$-statistics are also observed statistically significant at $P < .05$. The path coefficient of usability experts' opinion is found negative. Negative $t$-statistics and $P > .05$ make usability experts' opinion statistically insignificant in this research model. $R^2$ and adjusted $R^2$ of overall research model are observed as 0.294 and 0.259, respectively, with an $F$-ratio of 8.335 significant at $P < .05$.

## 6. Discussion

The use of open source software has increased in the recent years, mainly due to the easy access and availability of the Internet. Although it has been a common belief that OSS is popular with technically adept users, which results in a blurred boundary between its developers and users, the users of OSS are no more limited to this category alone: novice and nontechnical users are using OSS as well than ever before. As more and more people use OSS, usability and its related issues need to be addressed more seriously. Through empirical investigation, this research enables the OSS developers and project managers to realize the relationship of key factors of our research model and the OSS usability process. The results provide the empirical evidence and support for the theoretical foundations that the stated key factors play an important role in the institutionalization of usability within an OSS project.

Users' satisfaction plays a major role in the success of software, whether it is an open source or closed proprietary software. The more satisfied a target user is, particularly in application software, the more acceptability the software would get. And we believe that a path to achieve users' satisfaction goes through understanding their expectations and requirements. OSS is no longer a "*reserved arena*" for technically adept users; novice and nontechnical users from all over the world use open source software as well. As Koppelman and Van Dijk [22] identify that in order to know end-users' requirements and expectations, there is a need of more communication between the software developers and their target users, instead of relying on the former's instincts. 87% of our respondents support this observation that getting users' requirements helps in improving OSS usability. In our empirical investigation too, we have found a positive relationship between users' requirements and the OSS usability. *Users' requirements* could thus be taken by OSS developers' community as a key issue to improve usability of their projects.

Role of HCI and usability experts cannot be undermined in software development. This becomes more important in application software, where end-users are the direct audiences. In proprietary software development, particularly in big organizations, such experts are hired to have their valuable opinion to make their software more usable and acceptable to end-users. Considering voluntary nature of work and fewer resources in OSS development, we do not find such experts actively playing their role in OSS field. It might be because they do not find themselves "*welcomed into OSS projects*" as identified in [6]. Anyhow, our statistical findings do not significantly support the positive association of usability experts' opinion and OSS usability. In the parametric and nonparametric statistical analysis as well as in PLS and multiple regression testing, the results were not supported by a significant statistical level of confidence (refer to Tables 2, 3, and 4). Therefore, we conclude that our study has not been able to prove a positive association of *usability experts' opinion* and OSS usability.

Gradual and incremental introduction of advance features in software makes users feel more comfortable. It increases the acceptability and adaptability of the application. Yunwen and Kishida [29] advocate the modularized

system design, such that users encounter the difficulty levels gradually and progressively. Gaming softwares use incremental approach in their design all the time. Only after user completes one level, s/he is encouraged to move on to the more difficult levels. Using same approach in all designs can make software more accommodating for a common novice user. 69% of the respondents in our survey agree that gradual introduction of advance features in software would enhance its adaptability. Our research study has also found a positive impact of incremental design approach on OSS usability. We thus have considered *incremental design approach* as a key attribute towards improving OSS usability.

Software testing is an integral part of software life cycle. Holzinger [46] emphasizes the earlier usability testing in software life cycle and maintains that "*the earlier critical design flaws are detected, the more likely they can be corrected.*" However, being a subjective matter, software usability cannot be directly measured. Furthermore, difficulty being faced by users in reporting errors makes the situation worse. Nichols and Twidale [24] refer to such difficulties faced by the users in reporting usability bugs by stating "*Difficulties that a User May Experience with a Graphical User Interface May Not be Easy to Describe Textually*." 72% of the respondents in our survey agree that formal usability testing should be an integral part of software testing procedure. The findings of our empirical investigation also confirm a positive association between usability testing and OSS usability. We thus take *usability testing* as a key issue to improve usability of OSS projects.

Students of computer science and software engineering being the future software managers and developers need to understand the importance of usable systems more seriously. They should be encouraged to realize that coming up with a programming solution to a problem is not the ultimate goal; any system developed should meet users' expectations. The earlier they would incorporate the usability features in their designs, the better it would be for their projects, from maintenance point of view, too. We also have found a positive impact of knowledge of user-centered design methods on OSS usability, in our empirical investigation. Not a single respondent of our survey disagreed with our question that "*Computer Science/Software Engineering students (future software developers) must learn how to incorporate usability aspects in their software designs*." This could be a part of long-term solution to improve software usability and would be equally beneficial to both OSS and closed proprietary software organizations. We thus take *knowledge of UCD methods* as one of the key factors to improve usability of OSS projects.

*6.1. Limitations of the Study and Threats to External Validity.* Surveys, experiments, metrics, case studies, and field studies are examples of empirical methods used to investigate both software engineering processes and products [47]. Empirical investigations are subject to certain limitations which is the case of this study as well.

Threats to external validity are conditions that limit the researcher's ability to generalize the results of his/her experiment to industrial practice [48], which is the case with this study. Specific measures have been taken to support external validity; for example, a random sampling technique is used to select the respondent from the population in order to conduct experiments. We retrieve the data from the most active and well-known OSS reporting website, sourceforge.net, which has huge amount of projects listed.

The increased popularity of empirical methodology in software engineering has also raised concerns on the ethical issues [49, 50]. We have followed the recommended ethical principles to ensure that the empirical investigation conducted and reported here would not violate any form of recommended experimental ethics. Another aspect of validity is concerned with whether or not the study reports results that correspond to previous findings. First of all is the selection of independent variables in this work. We have used five independent variables to relate with the dependent variable of OSS usability. We realize that there could be other key factors that influence OSS usability, but we have kept the scope of this study within open source software as well as OSS developers' point of view. Some other contributing factors like OSS development culture, less resources of OSS projects as compared to resources of closed proprietary software projects developed in big organizations, voluntary involvement of developers in OSS projects, and so forth have not been considered in this study. Another limitation of this study is a relatively small sample size. Although we sent our survey to notable number of OSS developers subscribed to 18 different categories of software, we received only 106 responses. The relatively small sample size in terms of number of respondents has a potential threat to the external validity of this study. Although the proposed approach has some potential to threaten external validity, we have followed appropriate research procedures by conducting and reporting tests to improve the reliability and validity of the study, and certain measures were also taken to ensure the external validity.

## 7. Conclusion

In this paper, we empirically investigate the effect of key factors on OSS usability and find answer to the research question stated in this investigation. Results of this empirical investigation exhibit that the stated key factors of our research model assist in improving OSS usability. Empirical results of this study strongly support the hypotheses that *users' requirements, incremental design approach, usability testing, and knowledge of UCD methods* are positively associated with the usability of an OSS project. However we could not find any significant statistical support for *usability experts' opinion* on OSS usability.

The study conducted and reported here will enable OSS development teams to better understand the effectiveness of the relationships of the stated key factors and usability of their projects. The OSS developers need to take into consideration multiple key usability factors to improve usability aspect of software in general and their projects in particular. Currently we are working on to develop a maturity model

to assess the usability of open source software projects. This empirical investigation provides us some justification to consider these key factors as a measuring instrument. This study is one of the series of four studies that we are conducting in parallel, regarding OSS usability from users, contributors, and software industry's points of view.

## Appendix

### Key Usability Factors from OSS Developers' Point of View (Measuring Instrument)

*Users' Requirements:*

(1) users' requirements help in increasing software usability;

(2) understanding community expectations by the code contributors support the software usability;

(3) taking community feedback before and after formal release of every major version of software is vital in improving software usability;

(4) recording users' profile is crucial in understanding their requirements and expectations and hence supports OSS usability;

*Usability experts' Opinions (Usability experts are those personnale who have formal training and expertise in usability and HCI).*

(5) usability features can better be incorporated if usability experts' opinions are taken during every life-cycle phase;

(6) seeking usability experts' opinions will compromise freedom of OSS developers;

(7) OSS designs based on usability experts' opinions end up with GUI having standard usability norms but lacking innovation;

(8) usability experts' opinions are equally important and applicable for OSS as they are for closed proprietary software;

*Incremental Design Approach (Introduction of advanced features of software to users in an incremental way).*

(9) incremental increase in the difficulty level of software always makes user feel more comfortable;

(10) a novice user needs only basic features of software;

(11) gradual introduction of advance features will enhance adaptability of the software; however it is not always possible;

(12) every user should explore advance features of software gradually;

*Usability Testing.*

(13) formal usability testing should be an integral part of software testing process;

(14) although software success is dependent on users' response, usability-related bugs mostly reflect personal demands;

(15) I will fix the usability-related bug only if I am convinced that the reported bug is worth fixing;

(16) usability bugs reflect users' requirements and expectations; therefore they need to be fixed on priority;

*Knowledge of User-Centered Design Methods ("UCD processes focus on users through the planning, design and development of a product" [51]).*

(17) computer science/software engineering students (future software developers) must learn how to incorporate usability aspects in their software designs;

(18) designing of user friendly GUI is an art not every programmer can learn;

(19) CS/SE curriculum needs to be revised to implant importance of usercenteredness in software designs;

(20) poor usability of OSS systems is not due to lack of knowledge of user-centered design methods; it is because they are not implemented and systems are not designed with people in mind.

*OSS Usability ("The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions" [5]).*

(1) improving OSS usability will result in reducing the overall cost, bug reporting and defects of the software;

(2) one reason of poor OSS usability is because it is developed free; OSS designers should have some incentive (e.g., award or recognition) to look for to;

(3) successful software project means usable software with satisfied users;

(4) software having improved usability and adaptability for less technical and novice users will end up benefiting all users.

## References

[1] E. S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, Sebastopol, Calif, USA, 1999.

[2] A. G. Koru and J. Tian, "Defect handling in medium and large open source projects," *IEEE Software*, vol. 21, no. 4, pp. 54–61, 2004.

[3] B. Fitzgerald, "The transformation of open source software," *MIS Quarterly*, vol. 30, no. 3, pp. 587–598, 2006.

[4] E. S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, Sebastopol, Calif, USA, 2001.

[5] International Standard ISO/IEC 9126-1, *Software Engineering—Product Quality—Part 1: Quality Model*, 1st edition, 2001.

[6] D. M. Nichols and M. B. Twidale, "The usability of open source software," *First Monday*, vol. 8, no. 1, 2005.

[7] H. Hedberg, N. Iivari, M. Rajanen, and L. Harjumaa, "Assuring quality and usability in open source software development," in *Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development*, Washington, DC, USA, May 2007.

[8] N. Viorres, P. Xenofon, M. Stavrakis, E. Vlachogiannis, P. Koutsabasis, and J. Darzentas, "Major HCI challenges for open source software adoption and development," in *Proceedings of the 2nd International Conference on Online Communities and Social Computing (OCSC '07)*, D. Schuler, Ed., pp. 455–464, Beijing, China, July 2007.

[9] R. Glass, "Is open source software more reliable? An elusive answer," *The Software Practitioner*, vol. 11, no. 6, 2001.

[10] T. Koponen, "Life cycle of defects in open source software projects," in *Proceedings of the 2nd International Conference on Open Source Systems*, pp. 195–200, 2006.

[11] M. Aberdour, "Achieving quality in open-source software," *IEEE Software*, vol. 24, no. 1, pp. 58–64, 2007.

[12] P. Wayner, *Free for All*, HarperCollins, New York, NY, USA, 2000.

[13] K. Crowston and B. Scozzi, "Bug fixing practices within free/libre open source software development teams," *Journal of Database Management*, vol. 19, no. 2, pp. 1–30, 2008.

[14] D. Cubranic and K. Booth, "Coordinating open-source software development," in *Proceedings of the 8th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 61–69, 1999.

[15] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the Apache server," in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 263–272, June 2000.

[16] P. A. David, A. Waterman, and S. Arora, "FLOSS-US, the Free/Libre/Open Source Software Survey for 2003," http://www.stanford.edu/group/floss-us/report/FLOSS-US-Report.pdf FLOSS-US-Report.pdf.

[17] J. Lerner and J. Tirole, "Some simple economics of open source," *Journal of Industrial Economics*, vol. 50, no. 2, pp. 197–234, 2002.

[18] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.

[19] K. R. Lakhani and E. Von Hippel, "How open source software works: "free" user-to-user assistance," *Research Policy*, vol. 32, no. 6, pp. 923–943, 2003.

[20] J. West, "How open is open enough? Melding proprietary and open source platform strategies," *Research Policy*, vol. 32, no. 7, pp. 1259–1285, 2003.

[21] J. Feller and B. Fitzgerald, "A framework analysis of the open source software development paradigm," in *Proceedings of the 21st Annual International Conference on Information Systems*, pp. 58–69, Brisbane, Australia, 2000.

[22] H. Koppelman and B. Van Dijk, "Creating a realistic context for team projects in HCI," *SIGCSE Bulletin*, vol. 38, no. 3, pp. 58–62, 2006.

[23] E. Golden, B. E. John, and L. Bass, "The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 460–469, St. Louis, Mo, USA, May 2005.

[24] D. M. Nichols and M. B. Twidale, "Usability processes in open source projects," *Software Process Improvement and Practice*, vol. 11, no. 2, pp. 149–162, 2006.

[25] G. Çetin and M. Göktürk, "A measurement based framework for assessment of usability-centricness of open source software projects," in *Proceedings of the 4th International Conference on Signal Image Technology and Internet Based Systems (SITIS '08)*, pp. 585–592, December 2008.

[26] L. Zhao and F. P. Deek, "Exploratory inspection: a learning model for improving open source software usability," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI '06)*, 2006.

[27] Z. Hussain, W. Slany, and A. Holzinger, "Current state of agile user-centered design: a survey, HCI and usability for E-inclusion," in *Proceedings of the 5th Symposium of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society (USAB '09)*, vol. 5889 of *Lecture Notes in Computer Science*, pp. 416–427, Springer, Berlin, Germany, 2009.

[28] S. Pemberton, "Scratching someone else's itch: (why open source can't do usability)," *Interactions*, vol. 11, no. 1, p. 72, 2004.

[29] Y. Yunwen and K. Kishida, "Toward an understanding of the motivation of open source software developers," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 419–429, May 2003.

[30] A. Holzinger, C. Stickel, M. Fassold, and M. Ebner, "Seeing the system through the end users' eyes: shadow expert technique for evaluating the consistency of a learning management system, HCI and usability for E-inclusion," in *Proceedings of the 5th Symposium of of the Austrian Computer Society (USAB '09)*, vol. 5889 of *Lecture Notes in Computer Science*, pp. 178–192, Springer, Berlin, Germany, 2009.

[31] X. Faulkner and F. Culwin, "Integrating HCI and SE," *ACM SIGCSE Bulletin*, vol. 32, no. 3, pp. 61–64, 2000.

[32] M. B. Rosson, J. M. Carroll, and C. M. Rodi, "Case studies for teaching usability engineering," *ACM SIGCSE Bulletin*, vol. 36, no. 1, pp. 36–40, 2004.

[33] N. Markov, "An introduction to the UCD methodology in the current environment," CASCON Workshop Report, 2003.

[34] A. Holzinger, P. Sammer, and R. Hofmann-Wellenhof, "Mobile computing in medicine: designing mobile questionnaires for elderly and partially sighted people," in *Proceedings of the 10th International Conference on Computers Helping People with Special Needs (ICCHP '06)*, vol. 4061 of *Lecture Notes in Computer Science*, pp. 732–739, Springer, Berlin, Germany, 2006.

[35] L. J. Cronbach, "Coefficient alpha and the internal structure of tests," *Psychometrika*, vol. 16, no. 3, pp. 297–334, 1951.

[36] A. H. van de Ven and D. L. Ferry, *Measuring and Assessing Organizations*, John Wiley & Sons, New York, NY, USA, 1980.

[37] A. Osterhof, *Classroom Applications of Educational Measurement*, Prentice-Hall, Upper Saddle River, NJ, USA, 2001.

[38] D. T. Campbell and D. W. Fiske, "Convergent and discriminant validation by the multitrait-multimethod matrix," *Psychological Bulletin*, vol. 56, no. 2, pp. 81–105, 1959.

[39] A. L. Comrey and H. B. Lee, *A First Course in Factor Analysis*, Psychology Press, Hillsdale, NJ, USA, 2nd edition, 1992.

[40] H. F. Kaiser, "A second generation little jiffy," *Psychometrika*, vol. 35, no. 4, pp. 401–417, 1970.

[41] H. F. Kaiser, "The application of electronic computers to factor analysis," *Educational and Psychological Measurement*, vol. 20, pp. 141–151, 1960.

[42] J. Stevens, *Applied Multivariate Statistics for the Social Sciences*, L. Erlbaum Associates, Hillsdale, NJ, USA, 1986.

[43] C. Fornell and F. L. Bookstein, "Two structural equation models: LISREL and PLS applied to consumer exit-voice

theory," *Journal of Marketing Research*, vol. 19, pp. 440–452, 1982.

[44] K. Joreskog and H. Wold, *Systems under Indirect Observation: Causality, Structure and Prediction*, Elsevier, North Holland, The Netherlands, 1982.

[45] http://en.wikipedia.org/wiki/P-value.

[46] A. Holzinger, "Usability engineering methods for software developers," *Communications of the ACM*, vol. 48, no. 1, pp. 71–74, 2005.

[47] J. Singer and N. G. Vinson, "Ethical issues in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1171–1180, 2002.

[48] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*, Kluwer Academic Publishers, Norwell, Mass, USA, 2000.

[49] R. R. Faden, T. L. Beauchamp, and N. M. P. King, *A History and Theory of Informed Consent*, Oxford University Press, Oxford, UK, 1986.

[50] J. Katz, *Experimentation with Human Beings*, Russell Sage Foundation, New York, NY, USA, 1972.

[51] http://www.upassoc.org/usability_resources/about_usability/what_is_ucd.html.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 307391, 17 pages
doi:10.1155/2010/307391

*Research Article*

# A Quality Model for Conceptual Models of MDD Environments

**Beatriz Marín,[1] Giovanni Giachetti,[1] Oscar Pastor,[1] and Alain Abran[2]**

[1] *Centro de Investigación en Métodos de Producción de Software, Universidad Politécnica Valencia, Camino de Vera s/n, 46022 Valencia, Spain*
[2] *Department of Software Engineering & Information Technology, École de Technologie Supérieure, Université du Québec, 1100 Notre-Dame Ouest, Montréal QC, Canada H3C 1K3*

Correspondence should be addressed to Beatriz Marín, bmarin@pros.upv.es

Received 15 February 2010; Accepted 9 June 2010

Academic Editor: Giovanni Cantone



In Model-Driven Development (MDD) processes, models are key artifacts that are used as input for code generation. Therefore, it is very important to evaluate the quality of these input models in order to obtain high-quality software products. The detection of defects is a promising technique to evaluate software quality, which is emerging as a suitable alternative for MDD processes. The detection of defects in conceptual models is usually manually performed. However, since current MDD standards and technologies allow both the specification of metamodels to represent conceptual models and the implementation of model transformations to automate the generation of final software products, it is possible to automate defect detection from the defined conceptual models. This paper presents a quality model that not only encapsulates defect types that are related to conceptual models but also takes advantage of current standards in order to automate defect detection in MDD environments.

## 1. Introduction

Historically, software production methods and tools have a unique goal: *to produce high-quality software*. Since the goal of Model-Driven Development (MDD) methods is no different, MDD methods have emerged to take advantage of the benefits of using of models [1] to produce high-quality software. Model-Driven Technologies [2] attempt to separate business logic from platform technology in order to allow automatic generation of software through well-defined model transformations. In a software production process based on MDD technology, the conceptual models are key inputs in the process of code generation. Thus, the conceptual models must provide a holistic view of all the components of the final application (including the structure of the system, the behavior, the interaction between the users and the system, and the interaction among the components of the system) in order to be able to automatically generate the final application. Therefore, the evaluation of the quality of conceptual models is essential since this directly affects the quality of the generated software products.

To evaluate the quality of conceptual models, many proposals have been developed following different perspectives [3]. There are proposals that are based on theory [4], experience [5], the observation of defects in the conceptual models in order to induce quality characteristics [6], the evaluation of the quality characteristics defined in the ISO 9126 standard [7] in conceptual models by means of measures [8], a synthesis approach [9], and so forth. Taking into account the advantages and disadvantages of each development perspective for quality frameworks [3], *defect detection* is considered as a suitable approach because it provides a high level of empirical validity provided by the variety of conceptual models that are observed. However, it is interesting to note that this approach is not broadly used in the software engineering discipline, even though defect detection is the most common quality evaluation approach used by other disciplines such as health care [10].

To develop an effective quality assurance technique, it is necessary to know what kind of defects may occur in conceptual models related to MDD approaches. Currently, there are some approaches that detect defects in conceptual models (such as [11, 12]), which are mostly focused on

the detection of defects that come from either the data perspective (data models) or the process perspective (process models). However, defect detection has not been clearly accomplished from the interaction perspective (interaction models) even though all of these perspectives (data, process, and interaction modeling) are essential to specify a correct conceptual schema used in an MDD context [13].

In this article, we present an approach that allows the automatic verification of the conceptual models used in MDD environments with respect to defect types from the data, process, and interaction perspectives. We present a set of defect types related to data, process, and interaction models as well as a quality model that formalizes the elements involved in the identification of the different defect types proposed. This quality model, which is defined using current metamodeling standards, is the main contribution of this article. It is oriented to represent the abstract syntax of the constructs involved in the conceptual specification of software applications generated in MDD environments. From this quality model, defect detection can be automated by means of OCL [14] rules that verify the properties of the conceptual constructs according to the set of defects types defined.

The remainder of the paper is organized as follows. Section 2 presents the related work, including a list with the defect types found in the literature. Section 3 presents a set of conceptual constructs of the conceptual model of an MDD approach. Section 4 presents the quality model (metamodel) in detail. Section 5 presents our conclusions and suggestions for further work.

## 2. Related Work

In the literature, there is no consensus for the definition of the quality of conceptual models. There are several proposals that use different terminologies to refer to the same concepts. There are also many proposals that do not even define what they mean by quality of conceptual models. In order to achieve consensus about the definition of quality and to improve the conceptual models, we have adopted the definition proposed by Moody [3]. This definition is based on the definition of quality of a product or service in the ISO 9000 standard [15]. Therefore, we define the quality of a conceptual model as "*the total of features and characteristics of a conceptual model that bear on its ability to satisfy stated or implied needs*".

In order to design a quality model, the types of defects that the conceptual models used in MDD environments must be known. Defect detection refers to identifying anomalies in software products in order to correct them to be able to obtain better software products. The IEEE 1044 [16] presents a standard classification for software anomalies, which defines an anomaly as *any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, and so forth, or from someone's perceptions or experiences*. This definition is so broad that different persons can identify different anomalies in the same software artifact, and anomalies that one person identifies may not be perceived as anomalies for another person. Therefore, many researchers have had to redefine the concepts of error, defect, failure, fault, and so forth, while other researchers have used these concepts indistinctly [17]. To avoid the proliferation of concepts related to the software anomalies, in this article, we analyze the proposals of defect detection in conceptual models by adopting the terminology defined by Meyer in [18].

(i) *Error.* It is a wrong decision made during the development of a conceptual model.

(ii) *Defect.* It is a property of a conceptual model that may cause the model to depart from its intended behavior.

(iii) *Fault.* It is an event of a software system that departs from its intended behavior during one of its executions.

Taking into account that the cost of fault correction increases exponentially over the development life cycle [3], it is of paramount importance to discover faults as early as possible: this means detecting errors or defects before the implementation of the software system.

Travassos et al. [19] use reading techniques to perform software inspections in high-level, object-oriented designs. They use UML diagrams that are focused on data structure and behaviour. These authors advocate that the design artifacts (a set of well-related diagrams) should be read in order to determine whether they are consistent with each other and whether they adequately capture the requirements. Design defects occur when these conditions are not met. These authors use a defect taxonomy that is borrowed from requirements defects [20], which classifies the defects as Omission, Incorrect Fact, Inconsistency, Ambiguity, and Extraneous Information. However, they do not present the types of defects that were found in their study.

Laitenberger et al. [21] present a controlled experiment to compare the checklist-based reading (CBR) technique with the perspective-based reading (PBR) technique for defect detection in UML models. The authors define three inspection scenarios in the UML models in order to detect defects from different viewpoints (designers, testers, and programmers). These authors do not explicitly identify the types of defects found in UML models. However, they present a set of concepts that must be checked in the UML models from different viewpoints, and it is possible to infer the types of defects from these concepts.

Conradi et al. [22] present a controlled experiment that was designed to perform a comparison between an old reading technique used by the Ericsson company and an Object-Oriented Reading Technique (OORT) for detecting defects in UML models. The authors present a summary of the defects detected using both inspection techniques for the same project. The findings of the controlled experiment are that one group of subjects detected 25 defects using the old technique (without any overlaps of the defects detected) while the other group of subjects detected 39 defects using the OORT technique (with 8 overlaps in the defects detected). However, the authors did not present the types of defects found in the models inspected in the experiment.

Gomma and Wijesekera [23] present an approach for the identification and correction of inconsistency and incompleteness across UML views. It is applied in the COMET method [24], which uses the UML notation. The authors present 7 defect types related to the consistency between models: 1 defect type for the consistency between use-case diagrams and sequence diagrams, 4 defect types for the consistency between class diagrams and state transition diagrams, and 2 defect types for the consistency between sequence diagrams and state transition diagrams.

Kuzniarz [25] presents a set of inconsistencies found in student designs produced in a sample didactic development process. This proposal corresponds to a case study that was developed to explore the nature of inconsistency in UML designs. The authors present 8 defect types based on subjective but common-sense judgment.

Berenbach [26] presents a set of heuristics for analysis and design models that prevents the introduction of defects in the models. This allows semantically correct models to be developed. In addition, Berenbach presents the *Design Advisor* tool created by Siemens to facilitate the inspection of large models. This tool implements the heuristics proposed by Berenbach for evaluating the goodness of the analysis and design models. For the analysis models, Berenbach presents 10 heuristics for model organization, 5 heuristics for use case definition, 3 heuristics related to the use-case relationships, and 14 heuristics related to business object modeling. For the design models, he presents 2 heuristics for the class model.

Lange and Chaudron [27] identify the incompleteness of UML diagrams as a potential problem in the subsequent stages of a model-oriented software development. These authors refer to the completeness of a model by means of the aggregation of three characteristics: (1) the well-formedness of each diagram that comprises the model, (2) the consistency between the diagrams that comprise the model, and (3) the completeness among the diagrams that comprise the model. Note that the authors use the completeness concept to define the completeness of a model. Since this is equivalent to reusing the same concept (completeness) for its own definition, they do not really describe what is understood by completeness. These authors identify eleven types of defects of UML models: 5 types of defects related to the well-formedness of the diagrams, 3 types related to the consistency among the diagrams, and 3 other types related to the completeness among the diagrams.

Leung and Bolloju [28] present a study that aims to understand the defects frequently committed by novice analysts in the development of UML Class models. These authors use Lindland et al.'s quality framework [4] to evaluate the quality of the class diagrams. They distinguish five classifications that allow the evaluation of the syntactic quality, semantic quality, and pragmatic quality. These five classifications are *syntactic* (for the syntactic quality), *validity* and *completeness* (for the semantic quality), and *the expected is missing* and *the unexpected is present* (for the pragmatic quality). The authors obtain 103 different types of defects in 14 projects. However, the authors only detail 21 types of defects for one class diagram.

Bellur and Vallieswaran [12] perform an impact analysis of UML design models. This analysis evaluates the consistency of the design and the impact of a design change over the code. In order to evaluate the consistency of the design models, these authors propose evaluating the well-formedness of UML diagrams. The proposal of Bellur and Vallieswaran [12] extends the proposal of Lange and Chaudron [27] focusing on the quality of UML conceptual models as well as on the code. These authors identify 4 types of defects for use-case diagrams, 2 types of defects for sequence diagrams, 5 types of defects for the specification of the method sequences, 3 types of defects for the class diagram, 8 types of defects for the state transition diagrams, 2 types of defects for the component diagram, and 2 types of defects for the deployment diagram.

Summarizing, the above proposals present defect types that are related to the consistency (consistency is defined in the IEEE 610 standard as the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component) [29] among diagrams and defect types that are related to the correctness (correctness is defined in the IEEE 610 standard as the degree to which a system or component is free from faults in its specification, design, and implementation) [29] of a particular diagram. Table 1 presents the defect types of the different proposals analyzed. The first column, "Quality Characteristic", divides the defect types into two groups, the consistency characteristic and the correctness characteristic. The second column, "Authors", presents the authors who have proposed these defect types and the corresponding year of the proposals. The third column, "Model", presents the conceptual models (or diagrams) where the defect types have been found. The last column, "Defect Types", presents the defect types.

In the systematic revision of the state of the art, we noticed that all the proposals for defect detection in conceptual models are focused on UML models. However, it is well known that UML diagrams [30] do not have enough semantic precision to allow the specification of a complete and unambiguous software application [31–33], which is clearly observed in the semantic extension points that are defined in the UML specification [30]. For this reason, many methodologies have selected a subset of UML diagrams and conceptual constructs and have aggregated the needed semantic expressiveness in order to completely specify the final applications in the conceptual model, making the implementation of MDD technology a reality.

Since MDD proposals select a set of conceptual constructs and aggregate others to specify the conceptual models, it is important to note that a great number of conceptual constructs increase the complexity of the specification of the models and may cause the introduction of more defects into the conceptual models. For this reason, the conceptual constructs of an MDD proposal must be carefully selected so that the number of constructs that allow the complete specification of software applications at the conceptual abstraction level is as low as possible. In the following section, we present a minimal set of the conceptual constructs for an MDD environment.

Table 1: Defect types of conceptual models in the state of the art.

| Quality characteristic | Authors | Model | Defect types |
|---|---|---|---|
| Consistency among diagrams | Laitenberger et al. [21] | UML Class | (i) A class in the design class diagram is not a class in the system class diagram (with the same name) |
| | | | (ii) The number, types, and names of the attributes of a class in the design class diagram are not the same in the class of the system class diagram |
| | | | (iii) The number, names, and arguments of the methods of a class in the design class diagram are not the same in the class of the system class diagram |
| | | | (iv) The associations with their cardinality and arity in the design class diagram are not the same in the system class diagram |
| | | | (v) The constraints between classes of the design class diagram are not the same constraints for these classes in the system class diagram |
| | | UML Collaboration | (i) An object that does not correspond to a class of the class diagram |
| | | | (ii) The collaboration diagram has messages that do not correspond to the system operation |
| | | | (iii) The messages do not have the same number and type of arguments as the operations of the system described in the Operation model |
| | | Operation model of the Fusion method | (i) Operations (read, change, send, and result) that do not have the corresponding message in the collaboration diagram |
| | Gomma and Wijesekera [23] | UML Use-case | (i) A use case that does not correspond to at least one scenario described by an interaction diagram |
| | | COMET state transition | (i) Each Statechart that does not correspond to a state that is dependent on the control class in a class diagram |
| | | | (ii) The values of the current states, events, actions, and activities that appear on a Statechart that are not declared as attribute values of the respective state, event, action, and activity attributes for the state that is dependent on the control class |
| | | | (iii) An event on a Statechart that does not correspond to a method of the state of the control class in the class diagram |
| | | | (iv) Variables used to define conditions in any Statechart that are not attributes of the state that is dependent on the control class in the corresponding class diagram |
| | | | (v) Each event on a Statechart that corresponds to an incoming message on the state that is dependent on the control object, which is not represented in an interaction diagram (which executes the Statechart) |
| | | | (vi) Each action on a Statechart that corresponds to an outgoing message on the state that is dependent on the control object, which is not represented in an interaction diagram (which executes the Statechart) |
| | Kuzniarz [25] | UML Use-case | (i) The actor that is defined in the use case diagram is not the same actor that takes part in the interaction that is defined in the corresponding sequence diagram |
| | | | (ii) Not all the steps that are defined in the use case description correspond to messages in the system sequence diagram |
| | | | (iii) There are extension points for the extension of use cases that are missing (not represented) in the diagram of the controller use case |
| | | UML Sequence | (i) An iteration symbol that is related to an iterative task is missing in the sequence diagram |

Table 1: Continued.

| Quality characteristic | Authors | Model | Defect types |
|---|---|---|---|
| | | | (ii) There are links used in sequence diagrams that are not associations in the class diagram<br>(iii) There are sequences of messages in the sequence diagram that are not acceptable for the controller use case<br>(iv) There are elements used in pre- and postconditions of the contracts that are not defined in the class model<br>(vi) There are sequences of messages in the sequence diagram that are not an acceptable subsequence for the state machines that take part in the sequence diagram |
| | Berenbach [26] | UML Use-case | (i) Actors in use-case diagrams that are not specified in the context diagram<br>(ii) Use case without an interaction diagram that shows the possible scenarios |
| | | UML Class | (i) An interface in the class diagram that is not used to communicate with a concrete use case<br>(ii) A class that is not instantiated in any process of the system (sequence and collaboration diagrams)<br>(iii) Methods of the interface class that are not represented in the process of the system (sequence and collaboration diagrams)<br>(iv) Classes in the class diagram that are not specified in the use-case diagram<br>(v) Interfaces in the class diagram that are not specified in the use-case diagram |
| | Lange and Chaudron [27] | UML Sequence | (i) Messages between unrelated classes |
| | | UML Class | (i) Classes that are not called in the sequence diagram<br>(ii) Interfaces that are not called in the sequence diagram<br>(iii) Methods that are not called in the sequence diagram |
| | Lange and Chaudron [11] | UML Use-case | (i) Use cases without sequence diagrams |
| | | UML Sequence | (i) Objects of the sequence diagram that are not related to a class in the class diagram |
| | Bellur and Vallieswaran [12] | UML Use-case | (i) A use case that does not reference a use-case sequence diagram |
| | | UML Sequence | (i) A variable of a general class used in the sequence diagram that is null or is not a valid class in the class diagram<br>(ii) A method referenced in the sequence diagram that is null or is not a valid method in the method sequence charts<br>(iii) An object that is not the sender or the receiver in any interaction<br>(iv) An object that does not reference a valid class and state diagram<br>(v) A message that is not an instance of one class method for some class defined in the system |
| | | UML State Transition | (i) A state diagram that is not related to one and only one class<br>(ii) A state that is not described by one or more attributes of the class<br>(iii) State change events that do not correspond to messages in the method sequence diagrams |
| | | UML Component | (i) An intercomponent relationship that does not have 2 terminating end classes which are valid classes in the class diagram<br>(ii) A component in the component diagram that is not mapped to a physical system described in the deployment diagram |

Table 1: Continued.

| Quality characteristic | Authors | Model | Defect types |
|---|---|---|---|
| | | UML Deployment | (i) A deployment diagram that is not related to one or more component diagrams |
| Correctness | Laitenberger et al. [21] | UML Class | (i) The types of the attributes of a class are not specified<br>(ii) The methods of a class are not specified<br>(iii) Parameters that do not have a type associated |
| | Berenbach [26] | UML Use-case | (i) Multiple entry point for the system in the use-case diagram<br>(ii) Diagrams without a description and status<br>(iii) Concrete use cases without a definition<br>(iv) Abstract use cases that are not realized by a concrete use-case<br>(v) Extends use-case relationship that is specified between use cases that are not concrete<br>(vi) A concrete use case that includes an abstract use case |
| | | UML Business | (i) Services of business objects that do not have defined pre- and postconditions |
| | | UML Class | (i) An interface class with private methods |
| | Lange and Chaudron [27] | UML Sequence | (i) Objects without a name<br>(ii) Abstract classes in sequence diagrams<br>(iii) Messages without a name<br>(iv) Messages without a method |
| | | UML Class | (i) Classes without methods<br>(ii) Interfaces without methods<br>(iii) Classes with public attributes |
| | Leung and Bolloju [28] | UML Class | (i) Missing association label or cardinality detail<br>(ii) Improper label for a class, an association, an attribute, or an operation<br>(iii) Improper notation for an association, an aggregation, or a generalization<br>(iv) The nonimplicit operations that are present in sequence diagram are not included<br>(v) Implicit operation is listed<br>(vi) Wrong association cardinality (reversed or wrong range)<br>(vii) Wrong location of an attribute or an operation<br>(viii) Wrong association grouping<br>(ix) Missing class, attribute, operation, or association<br>(x) Incomplete class description<br>(xi) Operation that cannot be realized (using attributes and relationships)<br>(xii) Does not use domain-specific terminology<br>(xiii) Poor layout of the class diagram<br>(xiv) Insufficient distinction among sub-classes<br>(xv) Operation naming is improper or ambiguous<br>(xvi) Associations are replicated at sub-classes<br>(xvii) Manual operation is represented as association<br>(xviii) Excessive use of generalization, PK concept, FK concept, or emphasis on statistical information<br>(xix) Redundant attributes<br>(xx) Redundant associations<br>(xxi) Implementation detail is present in the diagram |

Table 1: Continued.

| Quality characteristic | Authors | Model | Defect types |
|---|---|---|---|
| | Lange and Chaudron [11] | UML Sequence | (i) Message in wrong direction in the sequence diagram |
| | | UML Class | (i) Multiple definitions of classes with equal names |
| | Bellur and Vallieswaran [12] | UML Use-case | (i) An actor that does not use one or more use cases<br>(ii) A use case that is not used by one or more actors<br>(iii) A use case that does not belong to system |
| | | UML Sequence | (i) A message that does not have a sender and a receiver object<br>(ii) A message that does not conform to the signature of the method corresponding to the message |
| | | UML Class | (i) A class diagram without classes<br>(ii) An association without a source and target class<br>(iii) A class that does not have at least one attribute or method |
| | | UML State Transition | (i) A state diagram without one start state and one end state<br>(ii) A state that has overlap of attribute values describing the state<br>(iii) A state that is not reachable from the start state<br>(iv) A state that cannot reach the end state |
| | | UML Component | (i) A component diagram without components |

## 3. Conceptual Constructs of an MDD Proposal

Model-Driven Development environments generally use Domain-Specific Modeling Languages (DSMLs), which define the set of conceptual constructs in order to represent the semantics of a particular domain in a precise way [34]. For example, the DSMLs for the Management Information Systems (MIS) domain share a well-known set of conceptual constructs. In order to allow the complete specification of MIS applications, the DSMLs must add specific conceptual constructs to the conceptual models.

The MDD methods use models at different levels of abstraction to automate their transformations to generate software products. Model-Driven Architecture (MDA) is a standard proposed by OMG [35] that defends MDD principles and proposes a technological architecture to construct MDD methods. This architecture divides the models into the following categories: Computation-Independent Models (CIMs), Platform-Independent Models (PIMs), and Platform-Specific Models (PSMs). CIMs are requirement models (e.g., use-case diagrams, i* models, etc.), which by nature do not allow the complete specification of final software applications. In contrast, PIMs and PSMs allow the complete specification of final applications in an abstract way, but the PSMs use constructs that are specific to the technological platform in which the final applications will be generated (e.g., java, c#, etc.). Therefore, in this article, we focus on the PIM models (which we refer to as conceptual models of MDD proposals) since they can be used independently of the platform.

To provide details of the conceptual constructs of MDD proposals for the MIS domain, we have selected a specific MDD environment as reference: the OO-Method approach [36, 37]. This approach is an object-oriented method that puts MDD technology into practice [36] by separating the business logic from the platform technology to allow the automatic generation of final applications by means of well-defined model transformations [37]. The OO-Method approach provides the semantic formalization that is needed to define complete conceptual models, which allows the specification of all the functionality of the final application at the conceptual level. The OO-Method approach has been implemented in an industrial tool [26] that automatically generates fully working applications. These applications can be either desktop or web MIS applications and can be generated in several technologies (java, C#, visual basic, etc.).

The conceptual model of an MDD proposal must be able to specify the structure, the behavior, and the interaction of the components of an MIS in an abstract way. Therefore, we distinguish three kinds of models that together provide the complete specification of software systems: a structural model, a behavior model, and an interaction model.

*3.1. The Structural Model.* The structural model describes the static part of the system and is generally represented by means of a class model. A class describes a set of objects that share the same specifications of characteristics, constraints, and semantics. A class can have attributes, integrity constraints, services, and relationships with other classes.

The attributes of a class represent characteristics of this class. The attributes of a class can also be derived attributes, which obtain their value from the values of other attributes or constants. The integrity constraints are expressions of a semantic condition that must be preserved in every valid state of an object.

Table 2: Defect types of conceptual models found using OOmCFP.

| Defect types found using OOmCFP |
|---|
| Defect: An object model without a specification of an agent class |
| Defect: An OO-Method Conceptual Model without a definition of the presentation model |
| Defect: A presentation model without the specification of one or more interaction units |
| Defect: An object model without the specifications of one or more classes |
| Defect: A class without a name |
| Defect: Classes with a repeated name |
| Defect: A class without the definition of one or more attributes |
| Defect: A class with attributes with repeated names |
| Defect: An instance interaction unit without a display pattern |
| Defect: A population interaction unit without a display pattern |
| Defect: A display pattern without attributes |
| Defect: Derived attributes without a derivation formula |
| Defect: A filer without a filter formula |
| Defect: An event of a class of the object diagram without valuations (excluding creation or destruction events) |
| Defect: A class without a creation event |
| Defect: Transactions without a specification of a sequence of services (service formula) |
| Defect: Operations without a specification of a sequence of services (service formula) |
| Defect: A service without arguments |
| Defect: A service with arguments with repeated names |
| Defect: A precondition without the specification of the precondition formula |
| Defect: A precondition without an error message |
| Defect: An integrity constraint without the specification of the integrity formula |
| Defect: An integrity constraint without an error message |

The services of a class are basic components that are associated with the specification of the behavior of a class. The services can be events, transactions, or operations. The events are indivisible atomic services that can assign a value to an attribute. The transactions are a sequence of events or other transactions that have two ways to end the execution: either all involved services are correctly executed or none of the services are executed. The operations are a sequence of events, transactions, or other operations, which are executed sequentially independently of whether or not the involved services have been executed correctly. The services can have preconditions that limit their execution because the preconditions are conditions that must be true for the execution of a service.

The relationships between classes in the structural model can be the following: agent, association, aggregation, composition, and specialization. Agents are relationships that indicate the classes that can access specific attributes or services of other classes of the model. Agents are specifically defined in the reference MDD approach.

*3.2. The Behavior Model.* The behavior model describes the dynamic part of a system. These dynamics include the behavior of each class and the interaction among the objects of the system. For the specification of the behavior of a system, we select the functional model of the reference MDD approach, which defines the behavior of the services defined inside the classes of the structural model.

The functional model specifies a formula with the sequence of events, transactions, or operations that must be executed when a service is used. This formula must be specified by means of well-formed, first-order logic formulae that are defined using the OASIS language [38].

The functional model specifies the effects that the execution of an event has over the value of the attributes of the class that owns the event. To do this, the functional model uses valuations to assign values to the corresponding attributes. The effect of a valuation is also specified using formulae within the syntax of the OASIS language. The change that a valuation produces in the value of an attribute is classified into three different categories: *state*, *cardinal*, and *situation*. The *state* category implies that the change of the value of an attribute depends only on the effect specified in the valuation for the event, and it does not depend on the value in the previous state. The *cardinal* category increases, decreases, or initializes the numeric-type attributes. The *situation* category implies that the valuation effect is applied only if the value of the attribute is equal to a predefined value specified as the current value of the attribute.

Since services can have preconditions, the conditions and the error messages of the preconditions are also specified using OASIS formulae. The integrity constraints of a class are also specified using OASIS formulae.

*3.3. The Interaction Model.* The interaction model describes the presentation (static aspects of a user interface like widgets, layout, contents, etc.) and the dialogs (dynamic

Table 3: 23 Defect types of conceptual models found using OOmCFP and OCL rules.

| Defect types found using OOmCFP | OCL rules |
|---|---|
| Defect: An object model without a specification of an agent class | **context** Agent **inv**: **body** self.allInstances->size()>0 |
| Defect: An OO-Method Conceptual Model without a definition of the presentation model | **context** ConceptualModel **inv**: **body** self.presentation->size()>0 |
| Defect: A presentation model without the specification of one or more interaction units | **context** PresentationModel **inv**: **body** self.interactionUnit->size()>0 |
| Defect: An object model without the specifications of one or more classes | **context** StructuralModel **inv**: **body** self.ownedClass->size()>0 |
| Defect: A class without a name | **context** Class **inv**: **body** Class.allInstances()->select(c \| c.name.isEmpty())->isEmpty() |
| Defect: Classes with a repeated name | **context** Class **inv**: **body** self.allInstances()->forAll(c1, c2 \| c1 < > c2 **implies** c1.name < > c2.name) |
| Defect: A class without the definition of one or more attributes | **context** Class **inv**: **body** self.features->select(t \| t.oclIsKindOf(TypedProperty))- >collect(t \| t.oclAsType (TypedProperty))->size()>0 |
| Defect: A class with attributes with repeated names | **context** Class **inv**: **body** self.features->select(t \| t.oclIsKindOf(TypedProperty))- >collect(t \| t.oclAsType (TypedProperty))->forAll(a1, a2 \| a1 < > a2 **implies a**1.name < > a2.name) |
| Defect: An instance interaction unit without a display pattern | **context** InstanceIU **inv**: **body** self.displaySet->size()>0 |
| Defect: A population interaction unit without a display pattern | **context** PopulationIU **inv**: **body** self.displaySet->size()>0 |
| Defect: A display pattern without attributes | **context** DisplaySet **inv**: **body** self.relatedAttribute->size()>0 |
| Defect: Derived attributes without a derivation formula | **context** DerivedAttribute **inv**: **body** self.derValue.effect->select(f \| f.value.isEmpty())->isEmpty() |
| Defect: A filter without a filter formula | **context** Filter **inv**: **body** self.filterFormula->select(f \| f.value.isEmpty())->isEmpty() |
| Defect: An event of a class of the object diagram without valuations (excluding creation or destruction events) | **context** Event **inv**: **body** self.allInstances->select(e \| (e.kind < > ServiceKind::creation and e.kind < > ServiceKind::destruction) implies e.valuation.size()> 0) |
| Defect: A class without a creation event | **context** Class **inv**: **body** self.features->select(s \| s.oclIsKindOf(Service))- >collect(s \| s.oclAsType (Service))-> select(s \| s.kind = ServiceKind::creation)->notEmpty() |
| Defect: Transactions without a specification of a sequence of services (service formula) | **context** Transaction **inv**: **body** self.effect->select(f \| f.value.isEmpty())->isEmpty() |
| Defect: Operations without a specification of a sequence of services (service formula) | **context** Operarion **inv**: **body** self.effect->select(f \| f.value.isEmpty())->isEmpty() |
| Defect: A service without arguments | **context** Service **inv**: **body** self.argument->size()>0 |
| Defect: A service with arguments with repeated names | **context** Service **inv**: **body** self.argument->forAll(a1, a2 \| a1 < > a2 **implies** a1.name < > a2.name) |
| Defect: A precondition without the specification of the precondition formula | **context** Service **inv**: **body** self.precondition.effect->select(f \| f.value.isEmpty())->isEmpty() |
| Defect: A precondition without an error message | **context** Service **inv**: **body** self.precondition->select (c \| c.errorMsg.isEmpty())->isEmpty() |

Table 3: Continued.

| Defect types found using OOmCFP | OCL rules |
|---|---|
| Defect: An integrity constraint without the specification of the integrity formula | **context** Constraint **inv**:<br>**body** self.effect->select(f \| f.value.isEmpty())->isEmpty() |
| Defect: An integrity constraint without an error message | **context** Constraint **inv**:<br>**body** self.allInstances->select (c \| c.errorMsg.isEmpty())->isEmpty() |

Table 4: 5 Defect types of conceptual models found in the literature and OCL rules.

| Defects Types found in the literature | OCL Rules |
|---|---|
| Defect: An attribute of a class without the specification of the type | **context** Class **inv**:<br>**body** self.features->select(t \| t.oclIsKindOf(TypedProperty))- >collect(t \| t.oclAsType (TypedProperty))->select(a \| a.type.isEmpty())->isEmpty() |
| Defect: An argument of a service without the specification of the type | **context** Service **inv**:<br>**body** self.features->select(s \| s.oclIsKindOf(Service))->collect(s \| s.oclAsType (Service)).argument.type->size<1->isEmpty() |
| Defect: Associations replicated at sub-classes | **Classifier**::parents(): Set(Classifier); parents = generalization.general<br>**Classifier**::allParents(): Set(Classifier); allParents = self.parents()->union(self.parents()->collect(p \| p.allParents())<br>**context** AssociationEnd **inv**:<br>**body** self.allInstances->forAll(r1, r2 \| r1.name = r2.name **and** r1.owningClass.allParents()->select(c \| c.name = r2.name)->isEmpty() |
| Defect: Associations with a repeated name | **Context** Relationship **inv**:<br>**body** self.allInstances()->forAll(r1, r2 \| r1 < > r2 **implies** r1.name < > r2.name) |
| Defect: An association without a source and target class | **context** Association **inv**:<br>**body** self.role->select(e1,e2 \| e1.role.kind = EndKind::source **and** e2.role.kind = EndKind::target) |

aspects of a user interface like controls, dynamic page change, etc.) of the system [39]. In order to specify the interaction between the users of an application and the system, the OO-Method MDD approach specifies views. A view corresponds to a set of interfaces, which are the communication point between agents and classes of the structural model. When the views of a system have been defined, the interaction model of each view must be specified.

The interaction model allows the specification of the graphical user interface of an application in an abstract way [40]. To do this, the interaction model has a set of abstract presentation patterns that are organized hierarchically in three levels: access structure, interaction units, and auxiliary patterns. The first level allows the specification of the system access structure. In this level, the set of entry options that each user of the application will have available is specified by means of a *Hierarchy Action Tree* (HAT).

Based on the menu-like view provided by the first level, the second level allows the specification of the interaction units of the system. The interaction units are groups of functionality that allow the users of the application to interact with the system. Thus, the interaction units of the interaction model represent entry-points for the application. These units can be the following.

(i) A *Service Interaction Unit* (SIU). This interaction unit represents the interaction between a user of the application and the execution of a system service. In other words, the SIUs allow the users of the application to enter the values for the arguments of a service and to execute the service. They also provide the users with the feedback of the results of the execution of the service.

(ii) A *Population Interaction Unit* (PIU). This interaction unit represents the interaction with the system that deals with the presentation of a set of instances of a class. In a PIU, an instance can be selected, and the corresponding set of actions and/or navigations for the selected instance are offered to the user.

(iii) An *Instance Interaction Unit* (IIU). This interaction unit represents the interaction with an object of the system. In an IIU, the corresponding set of actions and/or navigations for the instance are offered to the user.

(iv) A *Master Detail Interaction Unit* (MDIU). This interaction unit represents the interaction with the system through a composite interaction unit. An MDIU corresponds to the joining of a master interaction unit (which can be an IIU or a PIU) with a detail interaction unit (which can be a set of IIUs, PIUs, or SIUs).

The third level of the interaction model allows the specification of the auxiliary patterns that characterize lower level details about the behavior of the interaction units. These auxiliary patterns are the following.

(i) The *entry* pattern is used to indicate that the user can enter values for the arguments of the SIUs.

(ii) The *defined selection* pattern is used to specify a list of specific values to be selected by the user.

(iii) The *arguments dependency* pattern is used to define dependencies among the values of the arguments of a service. To do this, Event-Condition-Action (ECA) rules are defined for each argument of the service. The ECA rules have the following semantics: when an interface event occurs in an argument of a service (i.e., the user enters a value), an action is performed if a given condition is satisfied.

(iv) The *display set* pattern is used to specify which attributes of a class or its related classes will be shown to the user in a PIU or an IIU.

(v) The *order criteria* pattern allows the objects of a PIU to be ordered. This pattern consists of the ascendant/descendant order over the values of the attributes of the objects presented in the PIU.

(vi) The *action* pattern allows the execution of services by joining and activating the corresponding SIUs by means of actions.

(vii) The *navigation* pattern allows the navigation from an interaction unit to another interaction unit.

(viii) The *filter pattern* allows a restricted search of objects for a population interaction unit. A filter can have data-valued variables and object-valued variables. These variables can have a defined default value, an associated PIU to select the value of the object-valued variables, and precharge capabilities for the values of the object-valued arguments.

Each auxiliary pattern has its own scope that states the context in which it can be applied. With these conceptual constructs we can completely specify applications that correspond to the MIS domain in an abstract way. To formalize the concepts and the relationships among them, we present a generic metamodel for MDD proposals.

## 4. A Quality Model for MDD Environments

The quality model proposed in this article is specified by means of the modeling facilities that current metamodeling standards provide, specifically, the EMOF specification [41]. The EMOF was selected for its metamodeling language, which is supported by open-source tools such as the Eclipse EMF [42] (for metamodeling purposes) or Eclipse GMF [43] (for the generation of model editors). EMOF is also used by technologies such as ATL [44, 45] or QVT [46] for the implementation of model-to-model transformations. In the following subsections, we present the metamodel, the procedure proposed for defect detection, and a list of defect types found in MDD-oriented conceptual models.

*4.1. A Metamodel for Defect Detection in MDD-Oriented Conceptual Models.* In general terms, a metamodel is the artifact used to specify the abstract syntax of a modeling language: the structural definition of the involved conceptual constructs with their properties, the definition of relationships among the different constructs, and the definition of a set of rules to control the interaction among the different constructs specified [47]. In EMOF, a metamodel is represented by means of a class diagram, where each class of the diagram corresponds to a construct of the modeling language involved. We use the OCL specification [14] for the definition of the controlling rules of the metamodel since it is part of the OMG standards and can work with EMOF metamodels. Also, OCL rules provide a computable language for rule specification, which allows the defined rules to be automatically evaluated by existent tools such as Eclipse OCL tools [48].

With EMOF, the quality metamodel can specify the constructs involved in the different types of defects as well as the properties that must be present in the different conceptual constructs for the detection of defects. The OCL language used for the metamodeling rules can be used to define specific rules to automate defect detection. The final quality model is comprised of two main elements: (1) a metamodel for the description of the conceptual constructs that are used in MDD environments (which includes all the properties involved in defect detection) and (2) a set of OCL rules that allows the automatic detection of defects according to the list of defects presented in this article.

Figure 1 presents our quality metamodel. As this figure shows, a generic *ConceptualModel* of an MDD approach is comprised of a structural model (class *StructuralModel*), a behaviour model (class *BehaviourModel*), and an interaction model (*PresentationModel*). The structural model has a set of classes (class *Class*). Each class has several features (class *ClassFeature*), which can be services (class *Service*) or properties (class *Property*). In turn, the properties can be typed properties (class *TypedProperty*) or association ends (class *AssociationEnd*). The typed properties correspond to the attributes of a class, which must have a data type (class *DataType*) specified (attribute *Kind*). The typed properties can be derived (class *DerivedAttribute*) or not derived (class *Attribute*). The services can be events (class *Event*), transactions (class *Transaction*), or operations (class *Operation*). The events have valuations (class *Valuation*) to change the value of the attributes of a class. Each service has a set of arguments (class *Argument*) with their corresponding types (class *Type*), and it can also have a set of preconditions (association *precondition*). There are relationships between the classes of the model (represented by the class *RelationShip*), which can be associations (class *Association*), generalizations (class *Generalization*), and agents (class *Agent*). The agent definition is oriented to state the visibility and execution permissions over the classes of the defined model (association *agent*). The associations can be aggregations, compositions, or normal associations (attribute *aggregation* of the class *AssociationEnd*). Each class has a set of integrity constraints (association *integrityConstraint*). The classes, class features,

arguments, and relationships must have a name (class *NamedElement*).

The derived attributes, services, preconditions, and integrity constraints require the specification of the functionality that they perform. This functionality is specified by means of the behaviour model. The behaviour model has elements (class *BehaviourElement*) that can be conditional elements (*ConditionalBehaviourElement*) or constraint elements (*Constraint*). The conditional elements correspond to formulae (class *Formula*) with a condition (association *condition*) and an effect (association *effect*). The constraint elements correspond to formulae (class *Formula*) with an error message (attribute *errorMsg*). The formulae are defined (attribute *value*) by means of a particular language called OASIS, which is similar to the OCL language. Thus, the valuations and the specification of the derived attributes (class *ValueSpecification*) correspond to conditional behaviour elements, and the transactions, and operations correspond to behaviour elements. The preconditions and the integrity constraints correspond to constraint behaviour elements.

The interaction model has a set of interaction units (class *InteractionUnit*) and a set of auxiliary patterns (class *AuxPattern*) that allow the specification of the graphical user interface at an abstract level. The interaction units can be instances (*InstanceIU*), set of instances (*PopulationIU*), services (*ServiceIU*), and composite units (*MasterDetailIU*). The master-detail interaction units correspond to composite interaction units (class *DependentIU*), which are comprised of a master part (class *IndependentIU*) and a set of detail interaction units (class *DependentIU*). In the master part, only instances or populations can be used. In the detail part, instances, populations and other master detail interaction units can be used. Since, the instance interaction units and the population interaction units can be used independently of other interaction units; we classify them in the class *IndependentIU*. However, these interaction units can also be used inside the detail part of master detail interaction units, so we classify them in the class *DependentIU*. The independent interaction units have display sets (class *DiaplaySet*) to present the data. Each display pattern has a set of attributes (association *relatedattribute*) that are specified in the structural model, from which the data will be recovered to show the users of the application. The independent interaction units can have actions (class *ActionSet*) to present the set of services (throw a *ServiceIU*) that can be executed by the users over the instances shown in the interaction units. In addition, the independent interaction units can have navigations (class *NavigationSet*) to present the interaction units that can be accessed. The population interaction units can also have filters (class *Filter*) to search for information in a set of instances, which must be specified with the corresponding formula (class *Formula*). The service interaction units have entry (class *EntryPattern*) and selection patterns (class *SelectionPattern*), which have associated formulae composed of a condition (association *condition*) and an effect (association *effect*).

Since the quality metamodel has been specified using the standards of metamodelling, this metamodel eliminates redundancy of the elements defined and can be implemented using open-source modeling tools.

*4.2. The OOmCFP Procedure to Identify Defects.* Before the specification of the OCL rules, the types of defects that the conceptual models used in MDD environments must be known. To do this, we use a Functional Size Measurement (FSM) procedure designed for the OO-Method MDD approach, which is called OOmCFP [49]. This FSM procedure was developed to measure the functional size of the applications generated in an MDD environment from the conceptual models. The OOm CFP measurement procedure was defined in accordance with the COSMIC measurement manual version 3.0 [50]. Thus, a mapping between the concepts used in COSMIC and the concepts used in the conceptual model of the MDD approach has been defined [51].

The OOmCFP procedure is structured using three phases: the strategy phase, the mapping phase, and the measurement phase. The strategy phase addresses the four key parameters of software functional size measurement that must be considered before actually starting to measure: purpose of the measurement, scope of the measurement, identification of functional users, and level of granularity that should be measured. The mapping phase presents the rules to identify the functional processes, data groups, and data attributes in the software specification (i.e., in the conceptual model) depending on the parameters defined in the strategy phase. The measurement phase presents the rules to identify and measure the data movements that occur between the functional users and the functional processes.

OOmCFP starts with the definition of the strategy to perform the measurement. The *purpose* of the measurement in OOmCFP is defined as measuring the accurate functional size of the OO-Method applications generated in an MDD environment from the involved conceptual models. The *scope* of the measurement defines a set of functional user requirements that will be included in a measurement exercise. For OOmCFP, the scope of the measurement in OOmCFP is the OO-Method conceptual model, which is comprised of four models (Object, Dynamic, Functional, and Presentation), which allow a fully working software application to be generated.

Once the scope of the measurement has been determined, it is important to identify the *layers*, the *pieces of software*, and the *peer components* that make up the applications. Since the OO-Method software applications are generated according to a three-tier software architecture, we distinguish three *layers*: a client layer, which contains the graphical user interface; a server layer, which contains the business logic of the application; and a database layer, which contains the persistence of the applications (see Figure 2). In each layer of an OO-Method application, there is a *piece of software* that can interchange data with the pieces of software of the other layers. Thus, we distinguish, respectively, three pieces of software in an OO-Method application: the client piece of software, the server piece of software, and the database piece of software (see Figure 2).

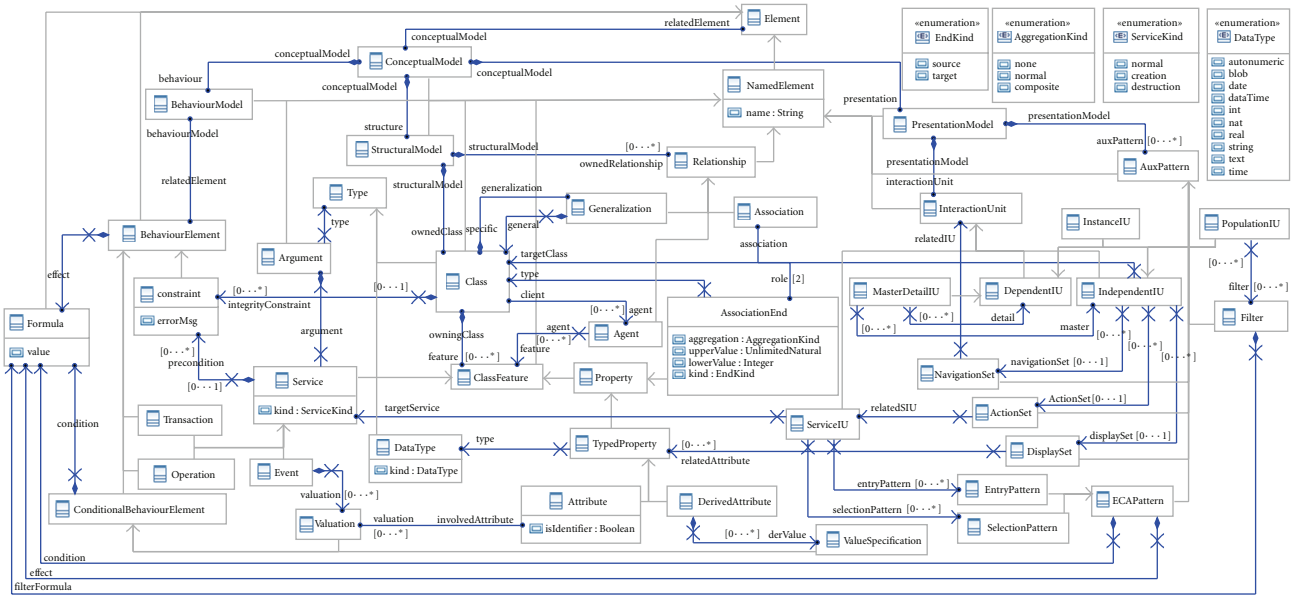


Figure 1: A metamodel for defect detection in MDD-oriented conceptual models.

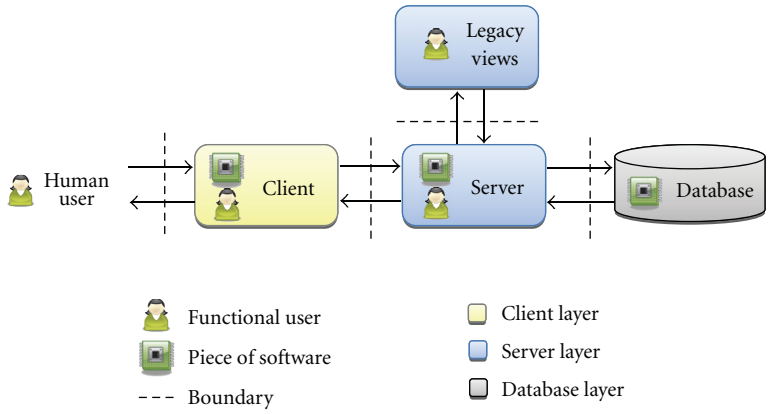


Figure 2: Layers, pieces of software, and functional users of the OO-Method applications.

Since the *functional users* are the types of users that send (or receive) data to (from) the functional processes of a piece of software, the functional users of the OO-Method applications are the human users, the client component of the software, the server component of the software, and the legacy views (see Figure 2). We called these users as "human functional user", "client functional user", "server functional user", and "legacy functional user", respectively.

Once the strategy is defined, OOmCFP starts a mapping phase. A *functional process* corresponds to a set of Functional User Requirements comprising a unique, cohesive, and independently executable set of data movements. A functional process starts with an entry data movement carried out by a functional user given that an event (*triggering event*) has happened. A functional process ends when all the data movements needed to generate the answer to this event have been executed. In the context of OOmCFP, the "human functional user" carries out the *triggering events* that occur in the real world. This functional user starts the *functional processes* that occur in the client layer of the application. In this layer, the functional processes are represented by the interaction units of the conceptual model that can be directly accessed by the "human functional user". The "client functional user" activates *triggering events* that occur in the interaction units of the presentation model. The "client functional user" starts *functional processes*, which are the actions that carry out the server layer of the software in response to the triggering events that occur in the client layer of the software. The "server functional user" carries out the *triggering events* that occur in the server layer of the software. The "server functional user" starts *functional processes*, which are the actions that the database layer carries out in response to the triggering events of the server layer, and the actions that the client layer carries out in response to triggering events of the server layer of the software. The "legacy functional user" activates *triggering events* that occur in the legacy system. The "legacy functional user" starts *functional processes*, which are the actions that the server layer

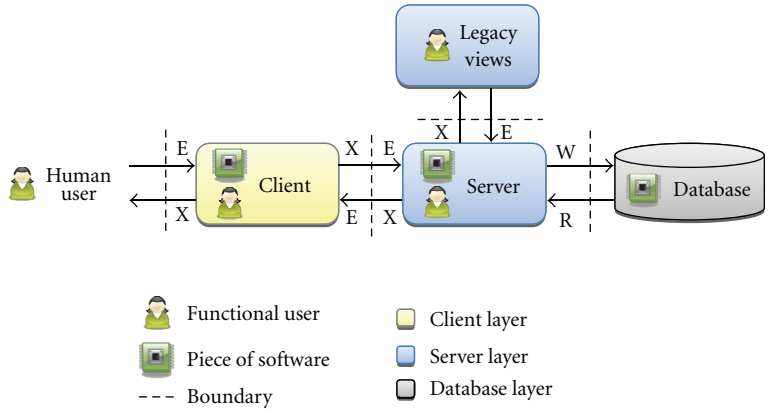


Figure 3: Data movements between the users and layers of an OO-Method application.

of the software carries out to interact with the legacy system. The *data groups* correspond to the classes of the structural model that participate in the functional processes. The *data attributes* correspond to the attributes of the classes identified as data groups.

In the measuring phase, the data movements correspond to the movements of data groups between the users and the functional processes. Each functional process has two or more data movements. Each data movement moves a single data group. A data movement can be an *Entry* (E), an *Exit* (X), a *Read* (R), or a *Write* (W) data movement. This proposal has 65 rules to identify the data movements that can occur in the OO-Method applications (see Figure 3). Each rule is structured with a concept of the COSMIC measurement method, a concept of the OO-Method approach, and the cardinalities that associate these concepts. These mapping rules detect the data movements (E, X, R, and W) of all the functionality needed for the correct operation of the generated application, which must be built by the developer of the application. This proposal has three measurement rules to obtain the functional size of each functional process of the application, each piece of software of the application, and the whole application. A complete description of OOmCFP can be found in its measurement guide (http://oomethod.dsic.upv.es/labs/images/OOmCFP/guide.pdf).

Since the OOmCFP procedure has been designed to obtain accurate measures of the applications that are generated from the OO-Method conceptual model [52] and has been automated to provide measurement results in a few minutes using minimal resources [53], we use it to verify the quality of conceptual models in three case studies. The OOmCFP measurement procedure assumes that the conceptual model is of high quality; that is, the OOmCFP procedure assumes that the conceptual model is consistent, correct, and complete. This is obviously an unreal assumption because conceptual models often have defects. Since a measurement procedure analyzes all the conceptual constructs that are related to the functionality of a system, we consider that a measurement procedure is a valuable tool for finding defects in conceptual models.

*4.3. Defect Types.* In order to determine the defect types of MDD conceptual models, the proposed metamodel and procedure were applied to three case studies with conceptual models of different functional sizes: a Publishing application (a small model), a Photography Agency application (a medium model), and an Expense Report application (a large model). We identified 39 defects and grouped them into 24 defect types (see [54]). For details, Table 2 shows the set of defect types that we identified using the OOmCFP procedure.

These defect types correspond to those related to structural models and those related to interaction models. This is one interesting contribution of our measurement procedure since, to our knowledge, there are no reported findings of defect types related to interaction models in the published literature. In order to formalize defect detection in the metamodel presented in Figure 1, we defined OCL rules to prevent the occurrence of the identified defects in the conceptual models. Table 3 shows the defect types and the OCL rules of our approach.

In order to identify the maximum number of defects using the quality model, we aggregated the defect types already found in the literature (see Table 1) with the corresponding OCL rules (see Table 4). We selected the defect types related to the class model, which is a diagram commonly used by several MDD proposals. We ruled out the defect types of the literature that were also identified using the OOmCFP measurement procedure.

In the three case studies mentioned above, the conceptual models did not achieve the characteristics of consistency and correctness due to the defect types presented in our approach. The OCL rules presented in Tables 3 and 4 can be implemented for the model compilers of MDD proposals in order to automatically verify the conceptual models with regard to these characteristics.

## 5. Conclusions

In this paper, we have presented a quality model to evaluate the conceptual models used in MDD environments

to generate final applications through well-defined model transformations. The quality model is comprised of the following: (1) a metamodel that contains a minimal set of conceptual constructs, their properties, and their relationships, which allows the complete specification of applications in the conceptual model of an MDD environment; and (2) a set of rules for the detection of defects in the model, which have been specified using OCL constraints [14].

The design of the metamodel has been systematically performed using an MDD approach as reference. This approach, which is known as OO-Method, has been successfully applied in the software industry. However, it is important to note that even though some elements of the metamodel are specific to the OO-Method MDD approach, equivalent modeling constructs can be found in other object-oriented MDD methods. Thus, this metamodel can be used as a reference to improve existent MDD approaches or as a starting point for the specification of new MDD-oriented modeling languages. Moreover, the main modeling constructs that compose the metamodel are the same constructs present in the UML specification. For this reason, the quality model presented here can be applied to other MDD methods that use UML-like models.

We take advantage of modeling, metamodeling, and transformation techniques to avoid having to manually identify defects in the conceptual models, which is an error-prone activity. Thus, the quality model (metamodel + OCL rules) has been designed for easy application to other MDD proposals. This is feasible because the EMOF standard is used to define the metamodel, which is supported by existent open-source tools [41, 48] and is also used by other MDD proposals for the specification of their modeling languages. Therefore, we can firmly state that the quality model proposed here contributes substantially to improving the MDD processes and the quality of software products generated in this context.

For future works, we plan to apply the quality model into different MDD approaches by using an integration process that automatically generates metamodeling extensions [55–57]. By using the integration proposal, we plan to show how the proposed quality model allows the automatic verification of the list of defect types found in MDD proposals. We also plan to develop empirical studies to evaluate the quality of conceptual models for different MDD proposals. The findings from this evaluation will be used to build a knowledge base for further improvements in the evaluation of the quality of conceptual models related to MDD approaches.

## Acknowledgement

This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and GVA ORCA PROMETEO/2009/015.

## References

[1] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003.

[2] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest editors' introduction: model-driven development,," *IEEE Software*, vol. 20, no. 5, pp. 14–18, 2003.

[3] D. L. Moody, "Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions," *Data & Knowledge Engineering*, vol. 55, no. 3, pp. 243–276, 2005.

[4] O. I. Lindland, G. Sindre, and A. Solvberg, "Understanding quality in conceptual modeling," *IEEE Software*, vol. 11, no. 2, pp. 42–49, 1994.

[5] T. H. Davenport and L. Prusak, *Working Knowledge: How Organisations Manage What They Know*, Business School Press, Boston, Mass, USA, 1998.

[6] W. L. Neuman, *Social Research Methods: Qualitative and Quantitative Approaches* , Needham Heights, Mass, USA, Allyn & Bacon, 4th edition, 2000.

[7] ISO/IEC, ISO/IEC 9126-1, Software Engineering—Product Quality—Part 1: Quality model,2001 .

[8] M. Genero, M. Piattini, and C. Calero, "A survey of metrics for UML class diagrams," *Journal of Object Technology*, vol. 4, no. 9, pp. 59–92, 2005.

[9] S. S.-S. Cherfi, J. Akoka, and I. Comyn-Wattiau, "Conceptual modeling quality—from EER to UML schemas evaluation," in *Proceedings of the 21st International Conference on Conceptual Modeling (ER '02)*, S. Spaccapietra, S. T. March, and Y. Kambayashi, Eds., vol. 2503 of *Lecture Notes in Computer Science*, Springer, Tampere, Finland, October 2002.

[10] R. M. Wilson, W. B. Runciman, R. W. Gibberd, B. T. Harrison, J. D. Newby, and J. D. Hamilton, "The quality in Australian health care study," *The Medical Journal of Australia*, vol. 163, no. 9, pp. 458–471, 1995.

[11] C. Lange and M. Chaudron, "Defects in industrial UML models—a multiple case study," in *Proceedings of the 2nd Workshop on Quality in Modeling of MODELS (QiM '07)*, pp. 50–79, Nashville, Tenn, USA, 2007.

[12] U. Bellur and V. Vallieswaran, "On OO design consistency in iterative development," in *Proceedings of the 3rd International Conference on Information Technology: New Generations (ITNG '06)*, pp. 46–51, IEEE, April 2006.

[13] J. Vanderdonckt, "Model-driven engineering of user interfaces: promises, successes, and failures," in *Proceedings of the 5th Annual Romanian Conference on Human-Computer Interaction (ROCHI '08)*, S. Buraga and I. Juvina, Eds., pp. 1–10, Matrix ROM, Iasi, Romania, 2008.

[14] OMG, Object Constraint Language 2.0 Specification, 2006.

[15] ISO, ISO Standard 9000-2000: Quality Management Systems: Fundamentals and Vocabulary, 2000.

[16] IEEE, IEEE 1044 Standard Classification for Software Anomalies, 1993.

[17] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[18] B. Meyer, *Object Oriented Software Construction*, Prentice Hall, New York, NY, USA, 2nd edition, 2000.

[19] G. H. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object oriented designs: using reading techniques to increase software quality," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, & Applications (OOPSLA '99)*, pp. 47–56, Denver, Colo, USA, October 1999.

[20] V. R. Basili, S. Green, O. Laitenberger et al., "The empirical investigation of perspective-based reading," *Empirical Software Engineering Journal*, vol. 1, no. 2, pp. 133–164, 1996.

[21] O. Laitenberger, C. Atkinson, M. Schlich, and K. E. Emam, "Experimental comparison of reading techniques for defect detection in UML design documents," *Journal of Systems & Software*, vol. 53, no. 2, pp. 183–204, 2000.

[22] R. Conradi, P. Mohagheghi, T. Arif, L. C. Hegde, G. A. Bunde, and A. Pedersen, "Object-oriented reading techniques for inspection of UML models—an industrial experiment," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '03)*, vol. 2749 of *Lecture Notes in Computer Science*, pp. 483–501, Springer, July 2003.

[23] H. Gomaa and D. Wijesekera, "Consistency in multiple-view UML models: a case study," in *Proceedings of the Workshop on Consistency Problems in UML-based Software Development II*, pp. 1–8, IEEE, San Francisco, Calif, USA, 2003.

[24] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, Reading, Mass, USA, 2000.

[25] L. Kuzniarz, "Inconsistencies in student designs," in *Proceedings of the Workshop on Consistency Problems in UML-Based Software Development II*, pp. 9–17, IEEE, San Francisco,Calif, USA, 2003.

[26] B. Berenbach, "The evaluation of large, complex UML analysis and design models," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 232–241, IEEE Computer Society, May 2004.

[27] C. Lange and M. Chaudron, "An empirical assessment of completeness in UML designs," in *Proceedings of the 8th Conference on Empirical Assessment in Software Engineering (EASE '04)*, pp. 111–121, IEEE, 2004.

[28] F. Leung and N. Bolloju, "Analyzing the quality of domain models developed by novice systems analysts," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pp. 1–7, January 2005.

[29] IEEE, IEEE 610 Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries, 1990.

[30] OMG, UML 2.1.2 Superstructure Specification, 2007.

[31] K. Berkenkötter, "Reliable UML models and profiles," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 203–220, 2008.

[32] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *IEEE Computer*, vol. 39, no. 2, pp. 59–66, 2006.

[33] A. L. Opdahl and B. Henderson-Sellers, "A unified modelling language without referential redundancy," *Data & Knowledge Engineering*, vol. 55, no. 3, pp. 277–300, 2005.

[34] G. Giachetti, B. Marín, and O. Pastor, "Perfiles UML y desarrollo dirigido por modelos: desafíos y soluciones para utilizar UML como lenguaje de modelado específico de dominio," in *Proceedings of the V Taller Sobre Desarrollo de Software Dirigido por Modelos (DSDM '09)*, Gijón, Spain, 2008.

[35] OMG, MDA Guide Version 1.0.1, 2003.

[36] O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano, "The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming," *Information Systems*, vol. 26, no. 7, pp. 507–534, 2001.

[37] O. Pastor and J. C. Molina, *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*, Springer, New York, NY, USA, 2007.

[38] O. Pastor, F. Hayes, and S. Bear, "OASIS: an object-oriented specification Language," in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE '92)*, pp. 348–363, Manchester, UK, 1992.

[39] J. Vanderdonckt, "A MDA-compliant environment for developing user interfaces of information systems," in *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAISE '05)*, O. Pastor and J. Falcão e Cunha, Eds., vol. 3520, pp. 16–31, Springer, Porto, Portugal, 2005.

[40] P. Molina, *Especificación de Interfaz de Usuario: De los Requisitos a la Generación Automática*, Universidad Politécnica de Valencia, Valencia, España, 2003.

[41] Eclipse Modeling Project, February 2010, http://www.eclipse.org/modeling/.

[42] Eclipse Modeling Framework Project, February 2010, http://www.eclipse.org/modeling/emf/.

[43] Eclipse Graphical Modeling Framework Project, February 2010, http://www.eclipse.org/gmf/.

[44] Eclipse ATL Project, February 2010, http://www.eclipse.org/m2m/atl/.

[45] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Proceedings of the Satellite Events at the MoDELS 2005 Conference*, vol. 3844 of *Lecture Notes in Computer Science*, pp. 128–138, Springer, 2006.

[46] OMG, QVT 1.0 Specification, 2008.

[47] B. Selic, "A systematic approach to domain-specific language design using UML," in *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '07)*, pp. 2–9, May 2007.

[48] Eclipse Model Development Tools, February 2010, http://www.eclipse.org/modeling/mdt/.

[49] B. Marín, N. Condori-Fernández, O. Pastor, and A. Abran, "Measuring the functional size of conceptual models in an MDA environment," in *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAISE '08)*, Z. Bellahsene, C. Woo, E. Hunt, X. Franch, and R. Coletta, Eds., pp. 33–36, Montpellier, France, 2008.

[50] A. Abran, J. Desharnais, A. Lesterhuis, et al., The COSMIC Functional Size Measurement Method—Version 3.0, 2007.

[51] B. Marín, N. Condori-Fernández, and O. Pastor, "Design of a functional size measurement procedure for a model-driven software development method," in *Proceedings of the 3rd Workshop on Quality in Modeling of MODELS (QiM '08)*, J.-L. Sourrouille, M. Staron, L. Kuzniarz, P. Mohagheghi, and L. Pareto, Eds., pp. 1–15, Touluse, France, 2008.

[52] B. Marín, O. Pastor, and A. Abran, "Towards an accurate functional size measurement procedure for conceptual models in an MDA environment," *Data & Knowledge Engineering*, vol. 69, no. 5, pp. 472–490, 2010.

[53] B. Marín, O. Pastor, and G. Giachetti, "Automating the measurement of functional size of conceptual models in an MDA environment," in *Proceedings of the Product-Focused Software Process Improvement (PROFES '08)*, vol. 5089 of *Lecture Notes in Computer Science*, pp. 215–229, Springer, 2008.

[54] B. Marín, G. Giachetti, and O. Pastor, "Applying a functional size measurement procedure for defect detection in MDD environments," in *Proceedings of the 16th European Conference on Systems & Software Process Improvement and Innovation (EUROSPI '09)*, R.V. O' Connor, Ed., vol. 42 of *Communications in Computer and Information Science*, pp. 57–68, Springer, Madrid, Spain, 2009.

[55] G. Giachetti, B. Marín, and O. Pastor, "Integration of domain-specific modelling languages and UML through UML profile extension mechanism," *International Journal of Computer Science & Applications*, vol. 6, no. 5, pp. 145–174, 2009.

[56] G. Giachetti, B. Marín, and O. Pastor, "Using UML as a domain-specific modeling language: a proposal for automatic generation of UML profiles," in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE '09)*, P. van Eck, J. Gordijn, and R. Wieringa, Eds., vol. 5565 of *Lecture Notes in Computer Science*, pp. 110–124, Springer, Amsterdam, The Netherlands, 2009.

[57] G. Giachetti, F. Valverde, and O. Pastor, "Improving automatic UML2 profile generation for MDA industrial development," in *Proceedings of the 4th International Workshop on Foundations and Practices of UML (FP-UML '08)*, vol. 5232 of *Lecture Notes in Computer Science*, pp. 113–122, Springer, 2008.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 729826, 21 pages
doi:10.1155/2010/729826

*Research Article*

# An Empirical Study of Social Networks Metrics in Object-Oriented Software

**Giulio Concas, Michele Marchesi, Alessandro Murgia, and Roberto Tonelli**

*Department of Electrical and Electronic Engineering, University of Cagliari, 09123 Cagliari, Italy*

Correspondence should be addressed to Roberto Tonelli, roberto.tonelli@dsf.unica.it

Received 14 April 2010; Accepted 13 August 2010

Academic Editor: Ewan Tempero



We study the application to object-oriented software of new metrics, derived from Social Network Analysis. Social Networks metrics, as for instance, the EGO metrics, allow to identify the role of each single node in the information flow through the network, being related to software modules and their dependencies. These metrics are compared with other traditional software metrics, like the Chidamber-Kemerer suite, and software graph metrics. We examine the empirical distributions of all the metrics, bugs included, across the software modules of several releases of two large Java systems, Eclipse and Netbeans. We provide analytical distribution functions suitable for describing and studying the observed distributions. We study also correlations among metrics and bugs. We found that the empirical distributions systematically show fat-tails for all the metrics. Moreover, the various metric distributions look very similar and consistent across all system releases and are also very similar in both the studied systems. These features appear to be typical properties of these software metrics.

## 1. Introduction

Measuring software to get information about its properties and quality is one of the main issues in modern software engineering. Limiting ourselves to object-oriented (OO) software, one of the first works dealing with this problem is the one by Chidamber and Kemerer (CK), who introduced the popular CK metrics suite for OO software systems [1]. Other OO metrics have been also proposed, like MOOD [2] and the Lorenz and Kidd metric suite [3], but the CK suite remains by far the most widely used. In fact, different empirical studies showed significant correlations between some of CK metrics and bug-proneness [4–7].

Modern software systems are made of many elementary units (software modules) interconnected in order to cooperate to perform specific tasks. In particular, in OO systems the units are the classes, which are in turn interconnected with each other by relationships like inheritance and dependency. Recently, it has been shown how these software systems may be analyzed using complex network theory [8–10]. In software networks, the classes are the nodes and the relationships among classes are the edges. This property opens the perspective to analyze software networks using metrics taken from other disciplines, like Social Network Analysis (SNA) [11]. These SNA metrics can be used together with more traditional product metrics, like class LOCS, number of Bugs, or the CK suite, to gain a deeper insight into the properties of software systems. Recent studies showed the importance of SNA metrics in measuring the interactions among software modules [12], and in particular how centrality measures are useful to identify software hubs, which show higher defect-proneness.

Considering software systems as graphs is not a new approach, and different authors have already investigated some of their properties, like the distribution of Fan-in or Fan-out of network nodes [8, 13], finding features characteristic of complex networks, like for instance, the presence of power-laws in the tail of the distributions of these metrics.

Only recently, SNA has been applied to the study of software systems. Zimmermann and Nagappan used SNA metrics to investigate a network of binary dependencies [12]. With regard to the study of OO software systems, only Tosun et al., to the authors' knowledge, applied SNA metrics to OO source code to assess defect prediction performance of these metrics [14]. In particular, there are no studies investigating

the relationships and the correlations among SNA metrics, traditional metrics, and Bugs metrics, and the corresponding statistical distributions. It must be noted that, when the measures are distributed according to power-laws, or other leptokurtotic distributions, traditional quantities like average or standard deviation may lose their meaning, and may not be characterizing measures anymore [15]. Knowledge of the overall statistical distribution is needed for characterizing the system properties. In particular, they are needed in order to obtain estimates of the metrics values for the future software releases.

In this paper, we study a set of releases of two large Open Source OO systems, Eclipse [16] and NetBeans [17], from the software network perspective, and compute the observed complementary cumulative distribution functions (CCDF) [15] of several metrics including SNA metrics. We study these systems because both the source code of several versions and complete data about Bugs and Issues of their software modules are available.

We study the relationships between these metrics and software fault-proneness—measured as the number of Bugs affecting software modules—and between them and more traditional software metrics. We also study the possibility of estimating the metric features for the future releases. For all the observed distributions, we performed best fits, finding analytical distributions able to model the system.

The systems analyzed are written in Java. All their classes are contained in Java source files, called Compilation Units (CU). A CU generally contains just one class, but less frequently it may contain two or more classes. We extracted the Bugs affecting files merging information found in bug-tracking repositories, specifically Bugzilla [18] and Issuezilla [19], with information taken from source code repositories, namely, Concurrent Versioning System (CVS) [20]. The information about Bugs and software changes (commit logs) is reported at CU level, and not at class level. Therefore, we extended the concept of software graph to CU level, building a graph in which nodes are Compilation Units and edges are the relationships between these CU's, extracted from the classes belonging to each CU. We used this graph for computing all the metrics analyzed as well as for computing the Bug distributions.

We found that most of the studied metrics are distributed according to the Yule-Simon distribution [21, 22], to a high degree of accuracy, and show a persistent or universal character across all different releases, for both systems analyzed. The high degree of accuracy of the analytical fitting distributions and their persistent character allowed us to estimate metrics values for the subsequent releases.

The paper is organized as follows. In Sections 2 and 3, we present related works and the research questions. In Section 4, we describe the software network obtained considering Compilation Units as nodes and redefine the CK metric suite for this case. Section 5 defines the SNA metrics analyzed. In Section 6, we discuss the process of Bug recovery and how to assign a Bug to the proper CU. In Section 7, we present the analytical distribution functions used to describe the empirical data and provide the main results. Section 8 analyses correlations between metrics and between metrics and Bugs. Section 9 shows how it is possible to use the results of previous sections to provide estimate for the feature of future releases. In Section 10, we discuss our finding and present the conclusions.

## 2. Related Work

Product metrics, extracted by analyzing static code of software, have been used to build models that relate these metrics to failure-proneness [4–7, 23]. Among these, the CK [24] suite is historically the most adopted and validated to analyze bug-proneness of software systems [4–7]. CK suite was adopted by practitioners [4] and is also incorporated into several industrial software development tools. Based on the study of eight medium-sized systems developed by students, Basili et al. [5] were among the first to find that OO metrics are correlated to defect density. Considering industry data from software developed in C++ and Java, Subramanyam and Krishnan [6] showed that CK metrics are significantly associated with defects. Among others, Gyimóthy et al. [7], studying an Open Source system, validated the usefulness of these metrics for fault-proneness prediction.

CK metrics are intended to measure the degree of coupling and cohesion of classes in OO software contexts. However, the studies using CK metrics do not consider the amount of "information" passing through a given module of the software network. Social Network Analysis (SNA) fills this gap, providing a set of metrics able to extract a new, different kind of information from software projects. Recently, this ability of SNA metrics was successfully employed to study software systems. Zimmermann and Nagappan [12] showed that network measures derived from dependency graphs are able to identify critical binaries of a complex system that are missed by complexity metrics. However, their results are obtained considering only one industrial product (Windows Server 2003). Tosun et al. [14] reproduced the previous work [12] extending the network analysis in order to validate and/or refute its results. They show that network metrics are important indicators of defective modules in large and complex systems. On the other hand, they argue that these metrics do not have significant effects on small scale projects. Both previous studies [12, 14] did not consider mutual relationships among SNA metrics and complexity metrics; therefore, they did not show if SNA metrics carry new information with respect to CK suite. Our work, instead, computes the correlation matrix among SNA metrics and CK metrics, considering also mutual correlations with respect to Issue, Bug, LOC, Fan-out and Fan-in.

## 3. Research Questions

The Pareto principle (80–20 rule) and the presence of power-laws in the tail of the distributions of many properties of software systems, including Bugs, have already been observed [9, 25, 26]. In [27], a high-order statistic coefficient was proposed to analyze software metrics exhibiting highly skewed statistical distributions, that was efficient in observing changes in software systems and in monitoring the development process.

We investigate if the new proposed SNA metrics possess the same properties and have similar empirical distributions. Moreover, the new metrics might possibly show correlations with Bugs and/or with other metrics and properties. Thus, it is desirable to study these correlations.

We also investigate if there are analytical distribution functions which may be used to describe such empirical distributions and possibly to forecast future properties of the software systems.

Consequently, our research questions are the following.

(i) RQ1: Are there analytical distribution functions describing the empirical data? Have these functions power-law behavior in their tails? What is the significance level of fitting empirical data with these distributions?

(ii) RQ2: Are these distributions similar in all the releases and in different systems, or tend to vary significantly?

(iii) RQ3: Is it possible to use these distributions to estimate the metrics values in subsequent releases?

(iv) RQ4: Are there SNA metrics significantly correlated with software Bugs, and to which extent?

(v) RQ5: Are there SNA metrics significantly correlated to traditional CK metrics, and to which extent?

## 4. CU Software Networks and CU-CK Metrics

An oriented graph can be associated to an OO software system, whose nodes are classes and interfaces, and whose edges are the relationships between classes, namely, inheritance, composition, and dependence. This approach has already been used in the literature. In [28] complex software networks were analyzed with nodes representing software entities at any level, and links representing syntactical relationships between modules, subprograms, and instructions. In [13] software is seen as a network of interconnected and cooperating components, choosing modules of varying size and functionalities, where the links connecting the modules are given by their dependencies. In [12] nodes are binaries, and edges are dependencies among binary pieces of code. In [29] interclass relationships were examined in three Java systems, and in [30] the same analysis was replicated on the source code of 56 Java applications. Object graphs were analyzed in [10] in order to reveal scale-free geometry of object-oriented programs, where the objects were the nodes and the links among objects were the network edges.

All these studies were devoted to exploit general dependencies among pieces of code in different software modules. With the same aim, in our study we do not distinguish between the various possibilities of software relationships, and with regard to SNA metrics, for simplicity, we do not even consider edges orientation, which would imply the construction of different EGO networks for the different kinds of links. Ours is a static analysis. Furthermore, since our software nodes are CUs, as explained later, many relationships among Java classes lose their original meaning at this granularity level. Our purpose is to focus on the role of the interactions among the software elements.

The number and orientation of edges allow to study the coupling between nodes, that is between classes. In this graph, the in-degree of a class, or Fan-in, is the number of edges directed toward the class. It measures how much this class is used by other classes of the system. The out-degree of a class, or Fan-out, is the number of edges leaving the class. It represents the level of usage the class makes of other classes in the system. Besides Fan-in and Fan-out metrics, we computed also, for each class, four CK metrics which were observed to be significantly correlated with the number of Bugs. They are as follows.

(i) Weighted Methods per Class (WMC). A weighted sum of all the methods defined in a class. We set the weighting factor to one, to simplify our analysis.

(ii) Coupling Between Objects (CBO). The counting of the number of classes which a given class is coupled to.

(iii) Response For a Class (RFC). The sum of the number of methods defined in the class, and the cardinality of the set of methods called by them and belonging to external classes.

(iv) Lack of Cohesion of Methods (LCOM). The difference between the number of noncohesive method pairs and the number of cohesive pairs.

We also computed the lines of code of the class (LOC), excluding blanks and comment lines. This is useful to keep track of the class size, because it is known that a "big" class is more difficult to maintain than a smaller class.

Every class is contained in a Java file, called CU. While most files include just one class, there are files including two or more classes. In Eclipse, about 10% of CUs host more than one class, whereas in Netbeans this percentage is about 30%.

While OO metrics and class graphs are usually referred to classes, Bugs and Issues are typically associated to CUs, because the logs of coding efforts aimed to fix Bugs are associated to changes to the source code, which are made to files (the CUs). Since the number of Bugs is of paramount importance to define software quality, to make Issue tracking consistent with source code we decided to base our analysis on CUs. Consequently, we extended CK metrics from classes to CUs. CUs represent therefore the main element of our study.

We defined a CU graph, whose nodes are the CUs of the system. Two nodes, $A$ and $B$, are connected with an edge directed from $A$ to $B$ if at least one class inside the CU represented by $A$ has a dependency relationship with one class inside the CU represented by $B$. Referring to this graph, we can compute In-links and Out-links of a CU-node. We reinterpreted LOC and CK metrics for this CU-graph:

(i) CU LOC is the sum of the LOCS of the classes contained in the CU;

(ii) CU CBO is the number of out-links of each node, excluding those representing inheritance. This definition is consistent with that of CBO metrics for classes;

(iii) CU LCOM and CU WMC are the sum of LCOM and WMC metrics of the classes contained in the CU, respectively;

(iv) CU RFC is the sum of weighted out-links of each node, each out-link being multiplied by the number of specific distinct relationships between classes belonging to the CUs connected to the related edge.

For each, CU we have thus a set of 7 metrics: In-links (Fan-in), Out-links (Fan-out), CU-LOCS, CU-LCOM, CU-WMC, CU-RFC, and CU-CBO. These metrics were computed for CUs of all versions of Eclipse and Netbeans.

## 5. SNA Metrics

Once the CU software graph is defined, we can compute on this graph the metrics used in Social Network Analysis. We restricted ourselves to the subset of SNA metrics that were found most correlated to software quality [12, 31]. Some of these metrics are the so-called "EGO metrics". For every node in the graph, there exists a subgraph composed by the node itself, called "EGO" (from the Latin word "ego", meaning "I"), and its immediate neighbors. Such subgraph is called the EGO Network associated to the node. The analysis of the EGO-networks gives information about the role of the "EGO" inside the entire network. In particular, EGO-network metrics provide insights on the extent each CU is connected to the entire system, and on the flow of information. In the definition of the EGO network, we considered the graph links as undirected links.

Other SNA metrics we considered, not directly related to the EGO network, are some centrality metrics, determining how important a given node/edge is relative to other nodes/edges in the network. Overall, we consider the following SNA metrics.

(i) Size: size of the EGO-network related to the considered node (i.e., Compilation Unit); it is the number of the nodes of the EGO-network.

(ii) Ties: number of edges of the EGO-network related to the node.

(iii) Brokerage: the number of pairs not directly connected in the EGO network, excluding the EGO node.

(iv) Eff-size: effective size of the EGO network; the number of nodes in the EGO network minus one, minus the average number of ties that each node has to other nodes of the EGO network.

(v) Nweak-comp: normalized Number of Weak Components; the number of disjoint sets of nodes in the EGO network without EGO node and the edges connected to it, divided by Size.

(vi) Reach-Efficiency; the percentage of nodes within two-step distance from a node, divided by Size.

(vii) Closeness; the sum of the lengths of the shortest paths from the node to all other nodes.

(viii) Information Centrality: the harmonic mean of the length of paths starting from all nodes of the network and ending at the node.

(ix) DwReach: the sum of all nodes of the network that can be reached from the node, each weighted by the inverse of its geodesic distance. The weights are thus 1/1, 1/2, 1/3, and so on.

All previous metrics are computed on the CU graph and are among those studied in [12]. It is useful to shortly describe how these SNA metrics may be relevant to software systems, namely, what they try to measure. The first five are strictly EGO metrics and describe the software neighborhood of a CU. Size measures the CU directly connected to a given CU while Ties, measured on such neighborhood, measures how dense are the software connections in this local network. Brokerage measures for how many couples of CU the given node acts like a broker, bridging the information flow among couples. Eff-size measures the redundancy of the connections in the EGO network, reducing the CU Size by an amount proportional to the local average Ties. If the average Ties is high, the local network has in fact redundant channels available for the information flow. The role of the EGO CU in the information exchange is then reduced. It must be noted that the average Ties refers only to the local network, and not to the global network, where, as we will see in the following, the distribution of Ties among all the nodes presents a fat tail. Nweak-comp measures how much the CU is needed to keep connected the other software units. The remainings are not EGO-metrics and are all centrality metrics. They measure if, in the global software network, the CU plays a peripheral rather than a central role.

We analyze the correlations among all of these metrics, as well as with the other metrics and with Bugs. For some metrics, we analyzed the statistical distributions and performed best fits with analytical distribution functions.

## 6. Issues Extraction

Bug Tracking Systems (BTSs) are commonly used to keep track of Bugs, enhancements, and features—called with the common term "Issues"—of software systems. The open source systems studied, Eclipse and Netbeans, make use of BTS Bugzilla and Issuezilla, respectively.

Each Issue inside a BTS is univocally identified by a positive integer number, the Issue-ID. BTS store, for each tracked Issue, its characteristics, life-cycle, software releases where it appears, and other data. In Bugzilla, a valid Bug is an Issue with a resolution of "fixed", a status of "closed", "resolved", or "verified", and a severity that is not "enhancement", as pointed out in Eaddy et al. [32]. Thus, Bugs are a subset of Issues. For Issuezilla, it is possible to adopt an equivalent definition: a Bug is an Issue with a resolution and status as above, and with type "defect".

Software configuration management systems like CVS (Concurrent Version System) keep track of all maintenance operations on software systems. These operations are recorded inside CVS in an unstructured way; it is not possible, for instance, on query CVS to know which operations were done to fix Bugs, or to introduce a new feature or enhancement. In order to identify Issues (Bugs)

Table 1: Number of CUs of Eclipse for each release.

| Release | 2.0 | 2.1 | 3.0 | 3.1 | 3.2 | 3.3 | 3.4 |
|---|---|---|---|---|---|---|---|
| Number of CU | 6391 | 7545 | 10288 | 11854 | 14138 | 15439 | 17387 |
| Release date | 06-2002 | 03-2003 | 06-2004 | 06-2005 | 06-2006 | 06-2007 | 05-2008 |

Table 2: Number of CUs of Netbeans for each release.

| Release | 3.2 | 3.3 | 3.4 | 4.0 | 6.0 | 6.1 |
|---|---|---|---|---|---|---|
| Number of CU | 3346 | 4383 | 6264 | 9317 | 31425 | 35034 |
| Release date | 04-2001 | 11-2001 | 08-2002 | 12-2004 | 12-2007 | 04-2008 |

affecting systems CUs, we had to match data stored in BTS with other data recorded in CVS of Eclipse and Netbeans.

All commit operations are committed to the CVS log messages as single entries. Each entry contains various data—among which the date, the developer who made the changes, a text message referring to the reasons of the commit, and the list of CU's interested by the commit. To obtain a correct mapping between Issue(s) and the related CU(s), the only way is to analyze the CVS log messages, to identify commits associated to maintenance operation where Issues are fixed. If a maintenance operation is done on a CU to address an Issue, we consider the CU as affected by this Issue.

In our approach, we first analyzed the text of commit messages, looking for Issue-IDs. In fact, in commit messages, there may be strings such as "Fixed 141181" or "bug no. 141181", but sometimes only the Issue-ID is reported. Unfortunately, every positive integer number is a potential Issue-ID, but sometimes numbers can refer to maintenance operations not related to Issue-ID resolution, such as branching, data, number of release, and copyright updating.

To avoid wrong mappings between Issue-IDs and CUs, we applied the following strategies.

(i) For each release, a CU can be hit only by Issues which are referred to in the BTS belonging to the same release.

(ii) We did not consider some numeric intervals particularly prone to host false positive Issue-IDs.

The latter condition is not particularly restrictive in our study, because we did not consider the first releases of the studied projects, where Issues with "low" ID appear. All IDs not filtered out are considered Issues and associated to the addition or modification of one ore more CUs, as reported in the commit logs. This method might not completely address the problems in the mapping between bugs and CUs [33]. In any case, we checked manually

- 10% of CU-bug(s) associations (randomly chosen) for each release,

- each CU-bug association for 6 subprojects (3 for Eclipse and 3 for Netbeans) without finding any error. A bias may still remain due to lack of information on CVS [33].

The total number of Issues affecting a CU in each release constitutes the Issue-metric we consider in this study, while the subset of Issues satisfying the conditions as in Eaddy et al. is the Bug-metric [32]. Clearly, not all source modules changed due to a Bug are to be considered "faulty". Some changes can happen to realign a correct piece of code with another piece of code that was modified to fix the Bug. So, what we measure is to what extent a Bug hits one, some, or many CUs, and not whether they were really faulty.

## 7. Empirical Results Regarding Metric Distributions

We systematically analyzed several main releases of Eclipse and Netbeans projects, namely, releases from 2.0 to 3.4 of Eclipse and releases from 3.2 to 6.1 of Netbeans. For each release, we computed the class graph and the consequent CU graph, and computed all the above quoted metrics at CU level. We analyzed the statistical distributions of the metrics among the systems CU's, which are our graph nodes, as well as the Bugs and Issues distributions. Note that we used CU metrics to be able to study more easily their relationships with Bugs and Issues. However, we verified that the behavior of CU metrics is absolutely similar to the behavior of the corresponding class metrics, for all considered metrics.

Tables 1 and 2 show the number of CUs in the various releases considered of Eclipse and Netbeans, respectively, together with their release date. Both the size and the release date of the considered systems vary considerably. The sizes—in number of CUs—vary of one order of magnitude in Netbeans, and about three times in Eclipse.

In the following figures, we systematically report the experimental CCDF (Complementary Cumulative Distribution Function) in log-log scale, as well as the best-fitting curves in many cases. This is convenient because, if the PDF (probability distribution function) has a power-law in the tail, the log-log plot displays a straight line for the raw data. This is a necessary but by no means a sufficient condition for power-law behavior. Thus we used log-log plots only for convenience of graphical representation, but all our calculations (CDF, CCDF, best fit procedures and the same analytical distribution functions we use) are always in normal scale.

The problems with representing the experimental PDF are that it is sensitive to the binning of the histogram used to calculate the frequencies of occurrence, and that bins with very few elements are very sensitive to statistical

noise. This causes a noisy spread of the points in the tail of the distribution, where the most interesting data lie. Furthermore, because of the binning, the information relative to each single data is lost. All these aspects make difficult to verify the power-law behavior in the tail. Thus, we adopted the CCDF representation, which presents various advantages. With this representation, there is no dependence on the binning, nor artificial statistical noise added to the tail of the data. If the PDF exhibits a power-law, so does the CCDF, with an exponent increased by one. Fitting the tail of the CCDF, or even the entire distribution, results in a major improvement in the quality of fit. An exhaustive discussion of all these issues may be found in [15].

We were able to obtain high quality best fits using three different distribution functions, all compatible with a power-law behavior in the tail. This approach has already been proposed in the literature to explain the power-law in the tail of various software properties [9, 13]. In our study, we are actually interested in answering our research questions, and the power-law behavior in the tail for the distribution of the studied metrics is only a side information, related to the best fitting distribution functions which may eventually best approximate the empirical data. With the best fitting proposed analytical distribution functions, we then try to predict some values for the metrics in the future releases.

The CCDF is defined as 1 − CDF, where the CDF (Cumulative Distribution Function) is the integral of the PDF. Denoting by $p(x)$ the probability distribution function, by $P(x)$ the CDF, and by $G(x)$ the CCDF, we have

$$\begin{aligned} G(x) &= 1 - P(x), \\ P(x) &= p(X \le x) = \int_{-\infty}^{x} p(x')dx' \\ G(x) &= p(X \ge x) = \int_{x}^{\infty} p(x')dx'. \end{aligned} \tag{1}$$

The distributions we study are a straight power-law—also called Pareto distribution—a log-normal, and a Yule-Simon distribution [21, 34]. The power-law is mathematically formulated as $p(x) \simeq x^{-\alpha}$, where $\alpha$ is the power-law exponent, the only parameter which characterizes the distribution, besides a normalization factor. Since for $\alpha \ge 1$ the function diverges in the origin, it cannot represent real data for its entire range of values. A lower cut-off, generally indicated $x_0$, has to be introduced, and the power-law holds above $x_0$. Thus, when fitting real data, this cut-off acts as a second parameter to be adjusted for best fitting purposes. Consequently, the data distribution is said to have a power-law in the tail, namely, above $x_0$.

The log-normal distribution has been also proposed in the literature to explain different software properties [13, 22, 30]. Mathematically it is expressed by

$$p(x) = \frac{1}{x\sqrt{2\pi\sigma^2}} e^{(\ln x - \mu)^2/2\sigma^2}. \tag{2}$$

It exhibits a quasi-power-law behavior for a range of values and provides high quality fits for data with power-law distribution with a final cut-off. Since in real data largest values are always limited and cannot actually tend to infinity, the log-normal is a very good candidate for fitting power-laws distributed data with a finite-size effect. Furthermore, it does not diverge for small values of the variable, and thus may also fit well the bulk of the distribution in the small values range.

The Yule-Simon distribution is expressed through the Euler Gamma function and has two parameters:

$$\begin{aligned} p(x) &= p_0 \frac{B(x+c,\alpha)}{B(h_0+c,\alpha)}, \\ B(a,b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}, \end{aligned} \tag{3}$$

where parameters $c$ and $\alpha$ are derived from the Yule model of the growth of Genera and Species in nature [15, 21, 34]. It produces a distribution with a power-law in the tail with exponent $\alpha$.

We started the analysis by computing the empirical CCDF's of the software network metrics for the various system studied. The empirical distributions of all considered SNA metrics show the same shape for all releases,both in Eclipse and Netbeans.Therefore,we show only the figures for some selected metrics for the last considered releases of the studied systems, namely, Eclipse-3.4 and Netbeans-6.0.

Figure 1 shows graph and SNA metrics for Eclipse 3.4. All CCDF are reported for convenience in log-log plots. Most CCDF show a small cut-off in the extreme tail, which is typically due to the finite size of the sample. Figure 2 shows the same data for Netbeans 6.0. The behavior of Netbeans metrics is very similar to Eclipse's, with smaller cut-off in the extreme tail, perhaps owing to the higher numbers of CUs.

In order to compare the empirical distributions across the releases, we show in the same plot two SNA metrics, Effective Size and Brokerage, for both Eclipse and Netbeans, to highlight their overlap. Figure 3 shows the persistence of the distributions of these metrics across three different releases, starting from the earliest to the most recent. In Eclipse, the curves slightly differ only in the tail, while in Netbeans they are almost coincident.

The empirical distributions of all considered metrics highly preserve the same shape, meaning that, for each specific metric, a single distribution function may account for the empirical data for all the system releases. Moreover, the distributions of the same metric look also very similar in Eclipse and Netbeans releases. Thus, once this distribution is known for one metric in one release, it is possible to infer the properties of the same metric in other releases, provided that the number of CUs is known.

Regarding what specific distribution function can best fit our empirical data, we experimented with the three distributions cited above—power-law, lognormal, and Yule Simon distributions. Figure 4 shows Fan-in, Fan-out, LOC, Size, and Ties, together with best-fit functions, for Eclipse-3.1. For the LOC metric, only the data with the Yule-Simon best-fit curve is shown, while for the other metrics data and best-fits with all the three distribution functions are shown in two different figures.

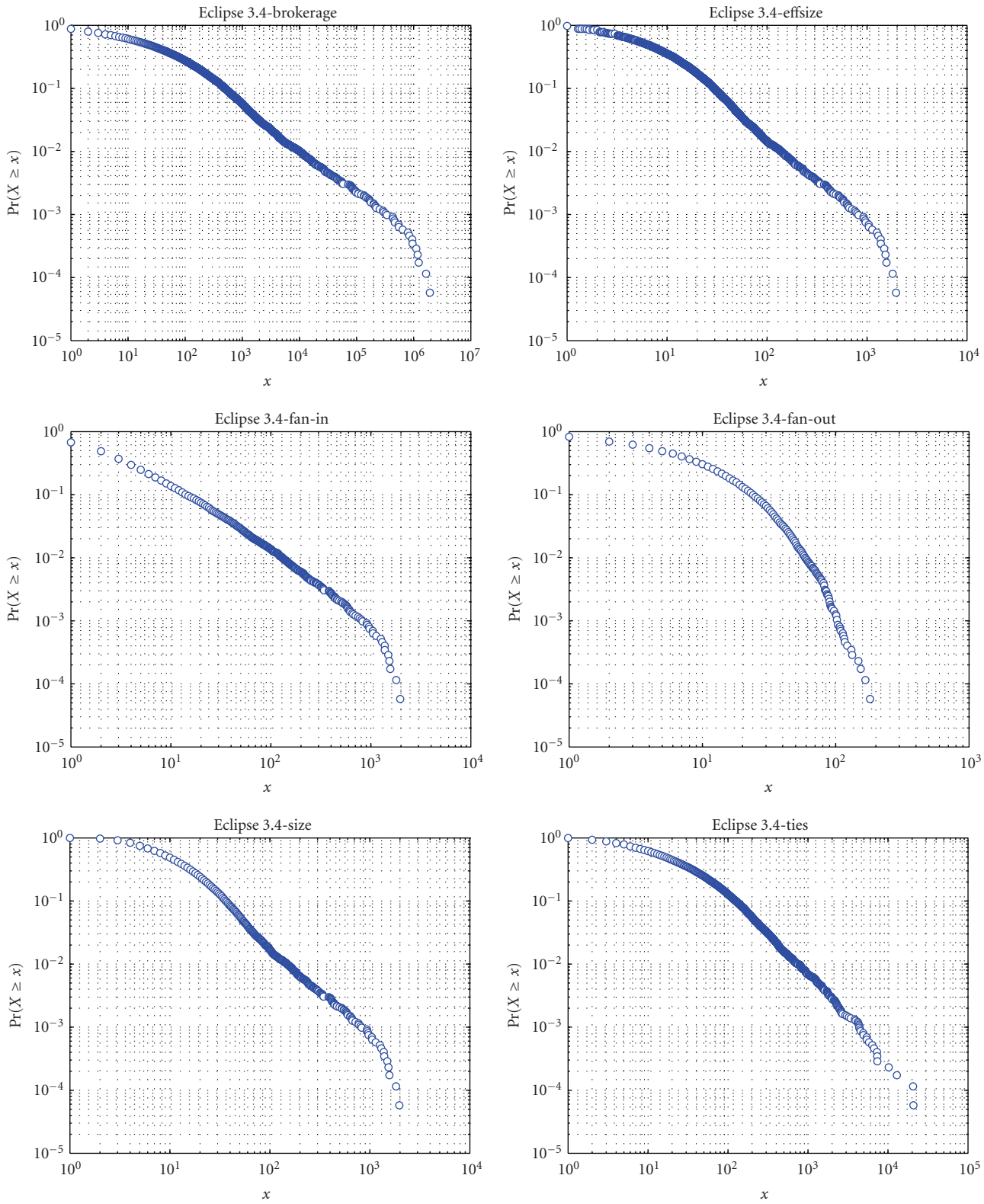


Figure 1: CCDF of SNA metrics for Eclipse 3.4 release. The name of the metrics is in the top of the box. The power-law behavior in the tail is patent for all metrics.

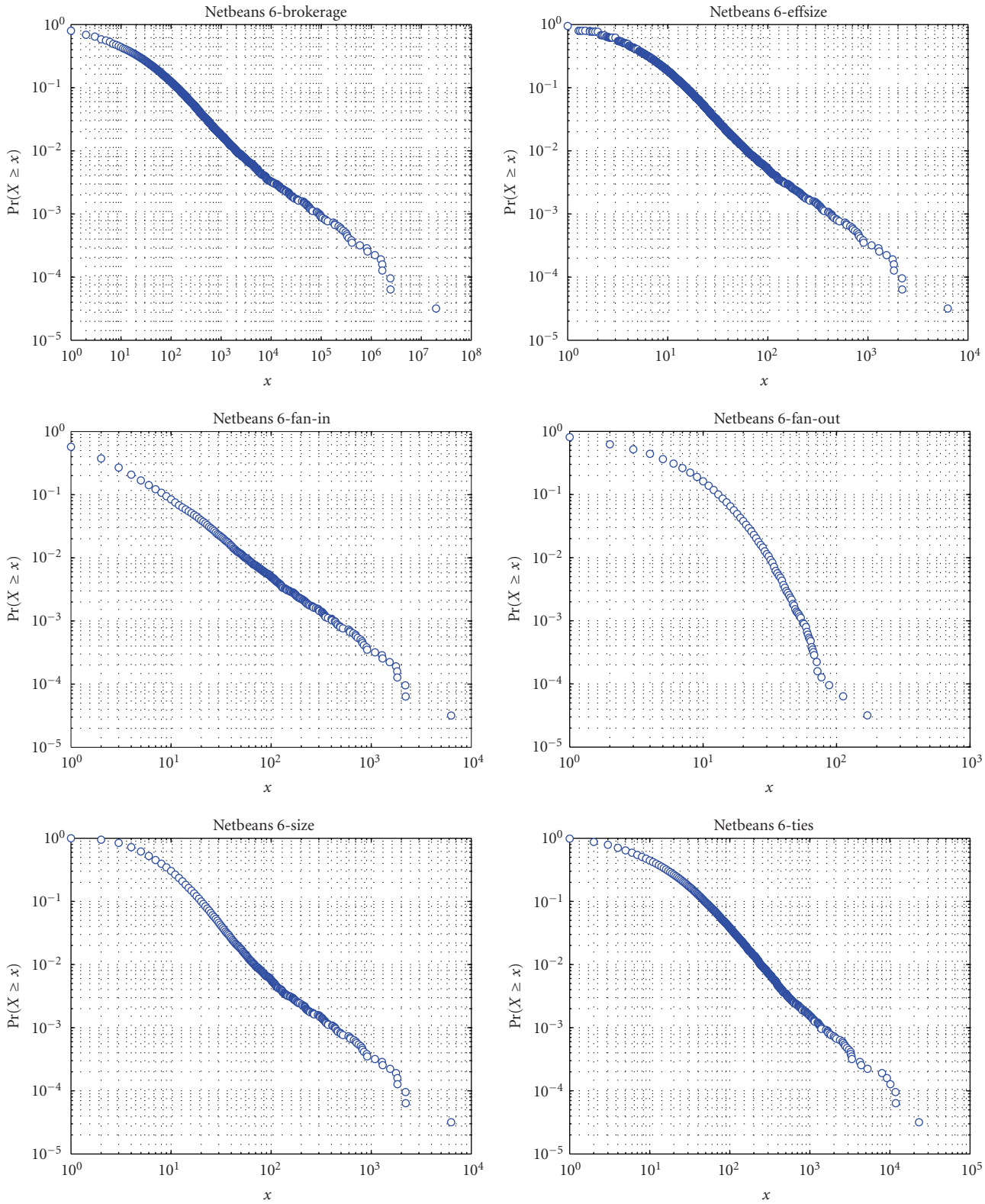


Figure 2: CCDF of SNA metrics for Netbeans 6.0 release. The name of the metrics is in the top of the box.

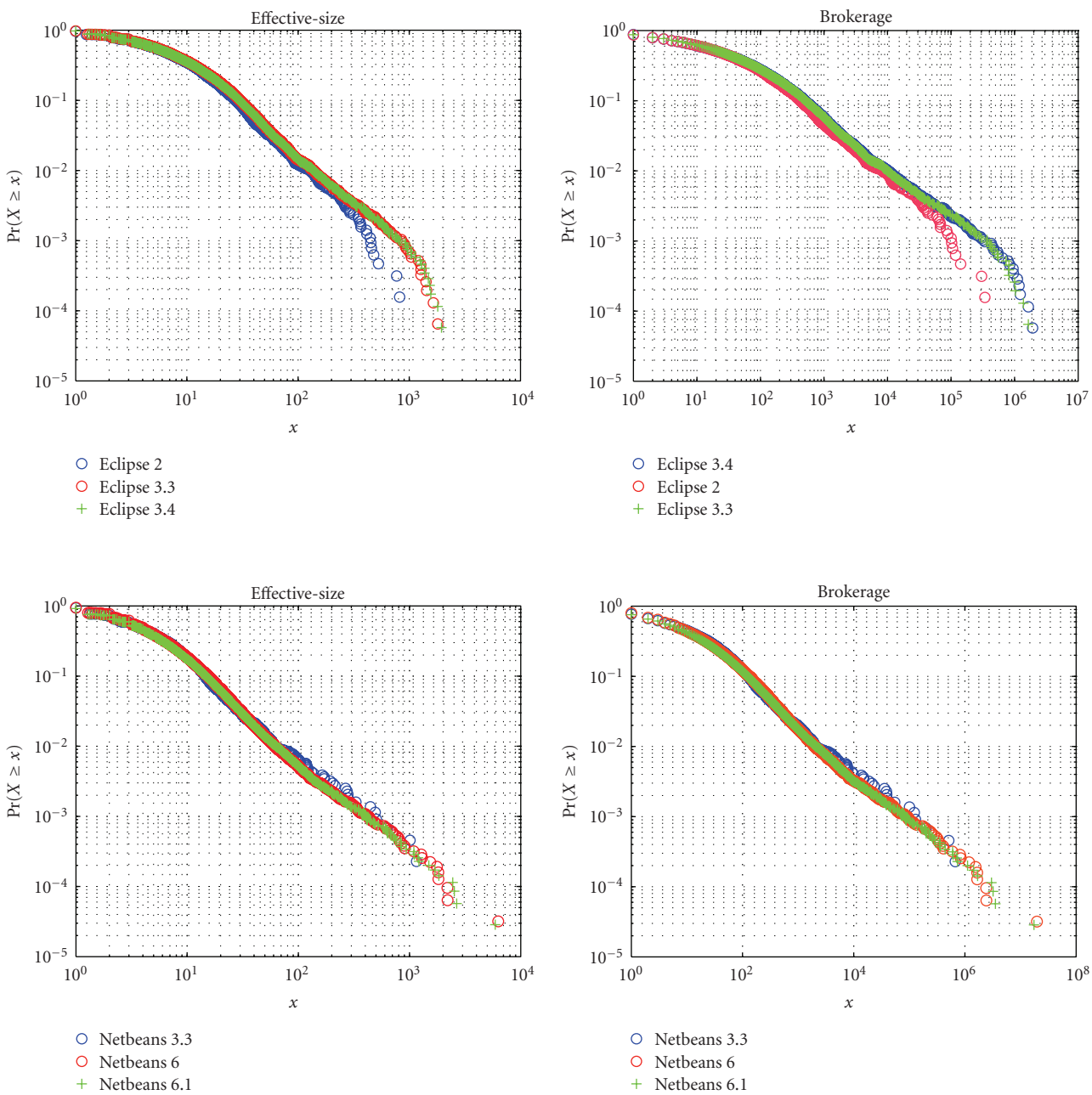


Figure 3: CCDF of EffSize and Brokerage metrics for various Eclipse and Netbeans releases. A very similar behavior is patent for all metrics and across all releases of the same system.

The fit using a truncated power-law is almost always very good. Note, however, that this fit is made starting from a minimum value $x_0$, denoting the value from which the power law tail is apparent. This makes easier to get good fits. The fit with a lognormal is usually the poorest.

This distribution is able to fit very well the bulk of the samples with small values, but in general it tends to zero too quickly with respect to empirical data. The fit with Yule-Simon distribution is sometimes very good, both for small values and in the tails. Other times, it fails to get a good fit in the tail.

In order to evaluate fit accuracy, we used the determination coefficient $R^2$, defined by $R^2 = 1 - \mathrm{SE}/\mathrm{ST}$, with

$$\mathrm{SE} = \sum_i (f_i - y_i)^2, \qquad \mathrm{ST} = \sum_i (\overline{y} - y_i)^2, \tag{4}$$

where $y_i$ are the empirical CCDF values and $f_i$ the corresponding best fitting values. All the fits have very high determination coefficients, sometimes up to 0.999 (Table 3).

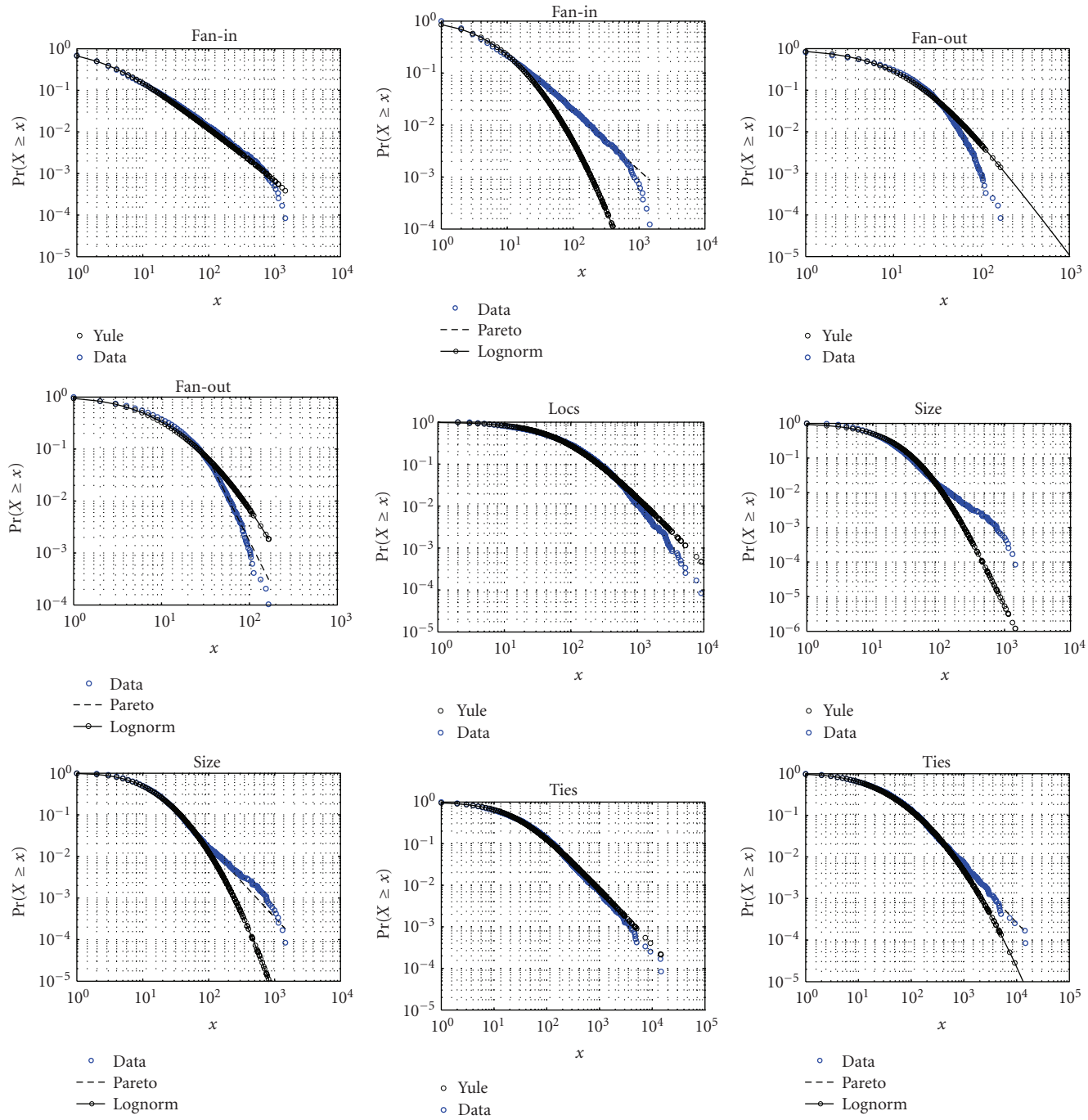


Figure 4: Empirical CCDFs of various metrics in Eclipse 3.1, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.

Table 3: Determination coefficients for the three distribution functions (Eclipse-3.1).

| $R^2$ | Yule-Simon | Lognormal | Power-law |
|---|---|---|---|
| Fan-in | 0.999 | 0.971 | 0.998 |
| Fan-out | 0.995 | 0.989 | 0.997 |
| Size | 0.987 | 0.999 | 0.998 |
| Ties | 0.998 | 0.999 | 0.999 |

Table 4: Determination coefficients for the three distribution functions (Netbeans-3.2).

| $R^2$ | Yule-Simon | Lognormal | Power-law |
|---|---|---|---|
| Fan-in | 0.999 | 0.978 | 0.998 |
| Fan-out | 0.998 | 0.982 | 0.996 |
| Size | 0.980 | 0.995 | 0.998 |
| Ties | 0.999 | 0.998 | 0.999 |

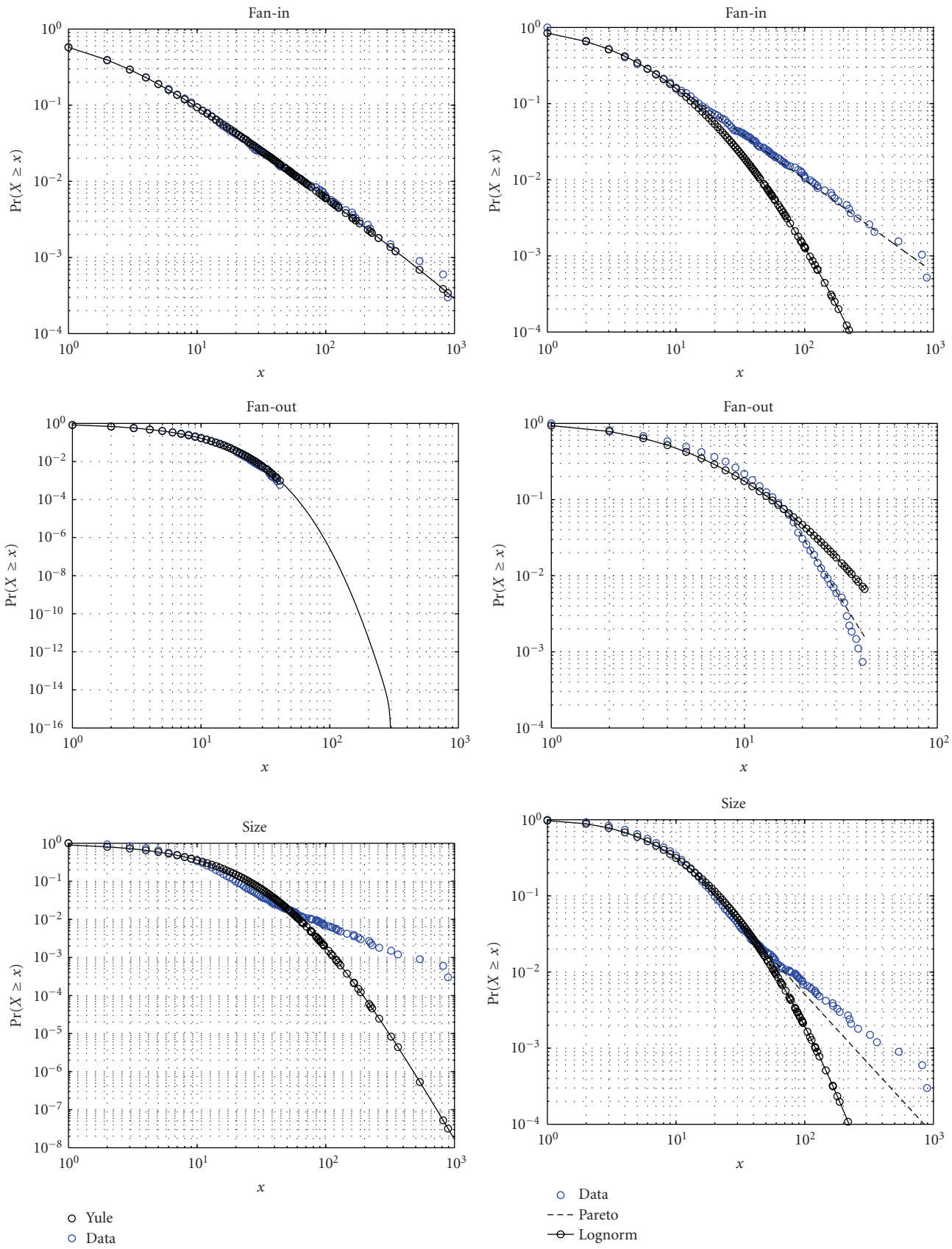


Figure 5: Continued.

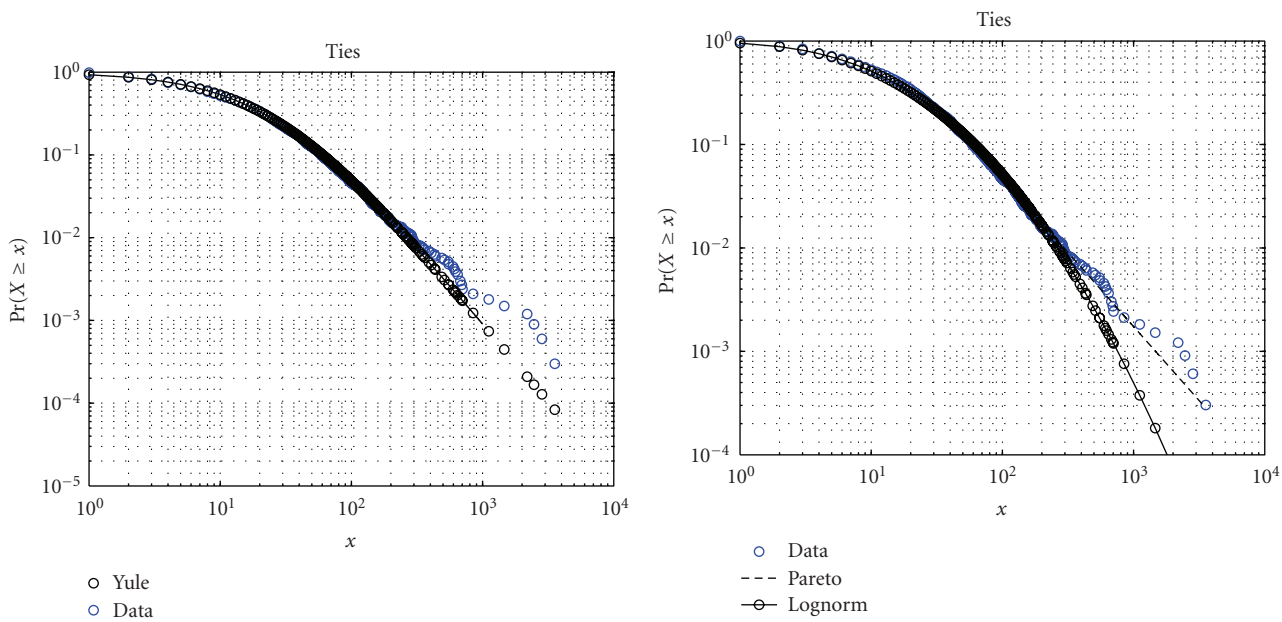


Figure 5: Empirical CCDFs of various metrics in Netbeans 3.2, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.

This suffices to answer to our research questions. It is in fact known that when experimental data are roughly power-law distributed, it is in general extremely difficult to assess the difference among a true power-law and other fat-tail distributions, since typically any statistical test does not rule out one or the other distribution function. In fact, they are often compatible with many different distribution functions [35].

Our purpose in this paper is, on the contrary, to provide a reasonable statistical description of the empirical data, and to find the analytical distribution function with the best fit. This allows us to make statistically reliable forecasts on the value assumed by some metrics in the future system releases. In our case, power-law is not in principle more interesting than the log-normal or Yule-Simon distributions, as long as these provide reliable estimates and good descriptions of the empirical data. Any other statistical speculation in order to discriminate among power-law or other distributions is out of our purposes.

Note that the determination coefficients are evaluated on the linear scale, whereas all the figures are in a log-log scale. In this scale, the discrepancy between best fitting curves and empirical curves is visually enhanced, especially in the tail, whereas in the original scale the fitting curves and the empirical ones visually overlap. On the other hand, our fitting procedure does not rely on any log-log representation of the data.

Figure 5 shows the corresponding data and best fits for Netbeans 3.2. Also for this system the curves provide a very good fitting of empirical data, for the various releases and for the different metrics. Again the coefficient of determination is always close to one (Table 4). The power-law provides an excellent approximation for the data in the tail above the $x_0$ cut-off, whose value depends on the metrics and on the system version.

The empirical studies presented above answer our first two research questions.

**R1**: *are there analytical distribution functions describing the empirical data? Have these functions power-law behavior in their tails? What is the significance level of fitting empirical data with these distributions?*

We definitely found that all studied metrics, traditional OO, network-based, and derived from Social Network Analysis, tend to follow precise analytical distributions to a high degree of significance level, according to our best-fitting criteria. These distributions are power-law—from a minimum value of data, $x_0$—lognormal and Yule-Simon distributions. All three distributions are compatible with a power-law behavior in their tail—regarding the lognormal distribution; this is true for datasets of finite size.

The fit using a truncated power-law are always very good. However, they depend on an *ad hoc* setting of the value $x_0$, and the power-law regards only the samples whose value $x \geq x_0$. Lognormal distribution shows good fits, according to the value of the determination coefficients, but not as good as power-law. Yule-Simon distribution, on the other hand, shows determination coefficients very similar to those of power-law, but the fit is over all the range of values. So, in general Yule-Simon distribution can be considered the best for most considered metrics.

**R2**: *are these distributions similar in all the releases and in different systems, or tend to vary significantly?*

We found that all considered metrics have a very consistent statistical behavior across all the releases of the same system, even when these releases span over years and have very different numbers of classes (and CUs).

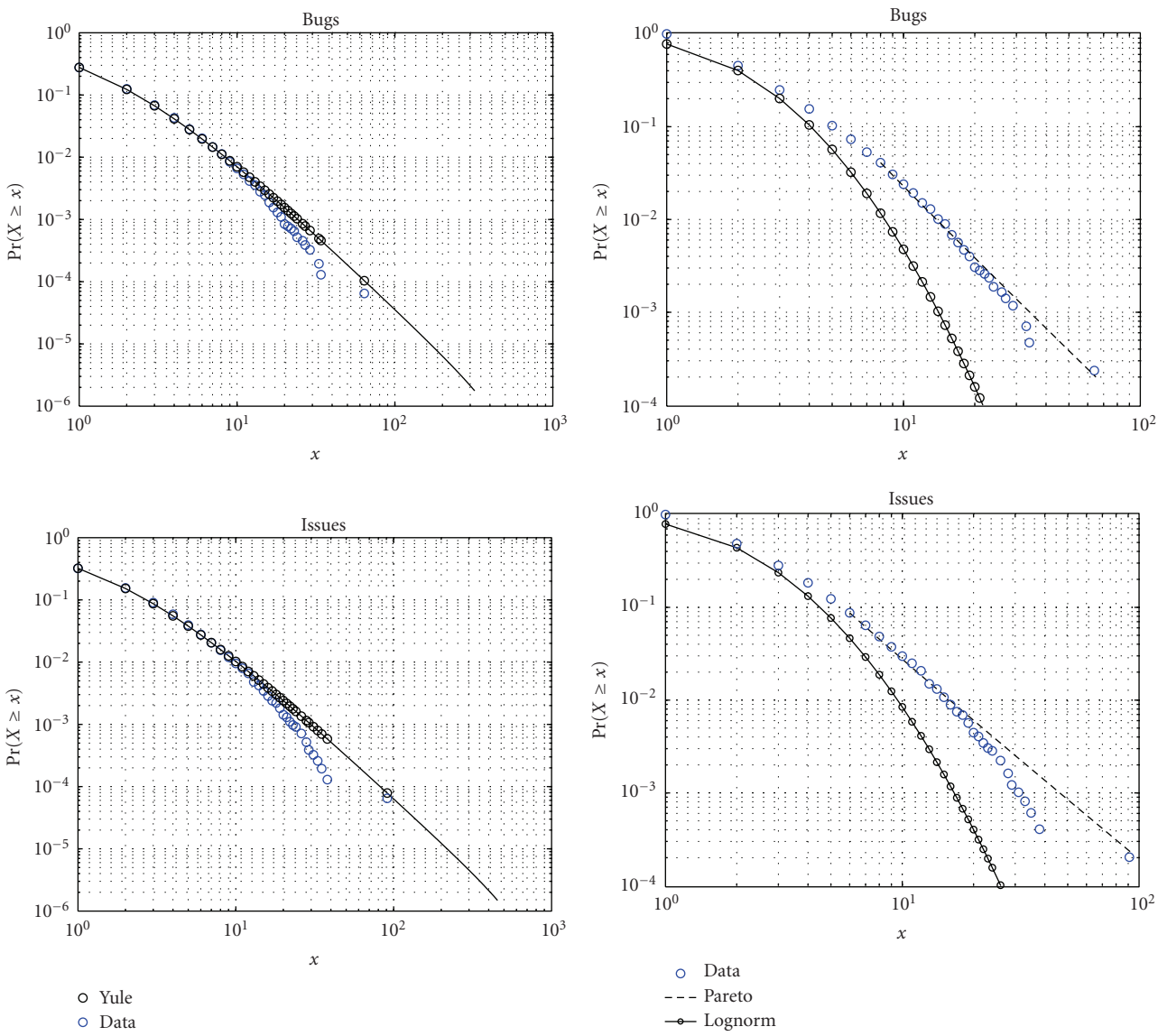


Figure 6: Empirical CCDFs of Bugs and Issues in Eclipse 3.3, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.

For completeness, we studied also other Java systems, belonging to the Qualitas Corpus [36] and found that the considered metrics, in systems with over one thousand classes, show behaviors very similar to those reported in this paper for Eclipse and Netbeans.

Next, we analyzed also the metrics related to Issues and Bugs. We found that also the distributions of Bugs and Issues follow similar patterns, in both Eclipse and Netbeans. In Figures 6 and 7, we show the empirical distributions of Issues and Bugs, for the releases 3.3 of Eclipse and 6.0 of Netbeans, together with the best fitting curves of the three considered distribution functions. All Issues and Bugs distributions are very similar throughout all Eclipse and Netbeans releases, so these figures can be considered typical.

The distributions of these metrics are well fitted by the simple power-law, according to the determination coefficient, above a threshold $x_0$, which depends on the particular data, and very well fitted by the Yule-Simon distribution since the beginning of the data. The log-normal distribution provides a worse fit, even if the determination coefficients $R^2$ are always above 0.94. Note again that the log-log scale enhances visually the distances in the tail, but the absolute values of the difference among fitting curves and empirical distributions are very small.

## 8. Correlations

In this section, we report the correlations among SNA metrics, CK metrics, and Bugs. Since the empirical distributions of all metrics are strongly not normal, correlations are better described using the Spearman coefficient. In our study, we computed also Pearson correlations, which are reported only in one case, for comparison. Our considerations, however,

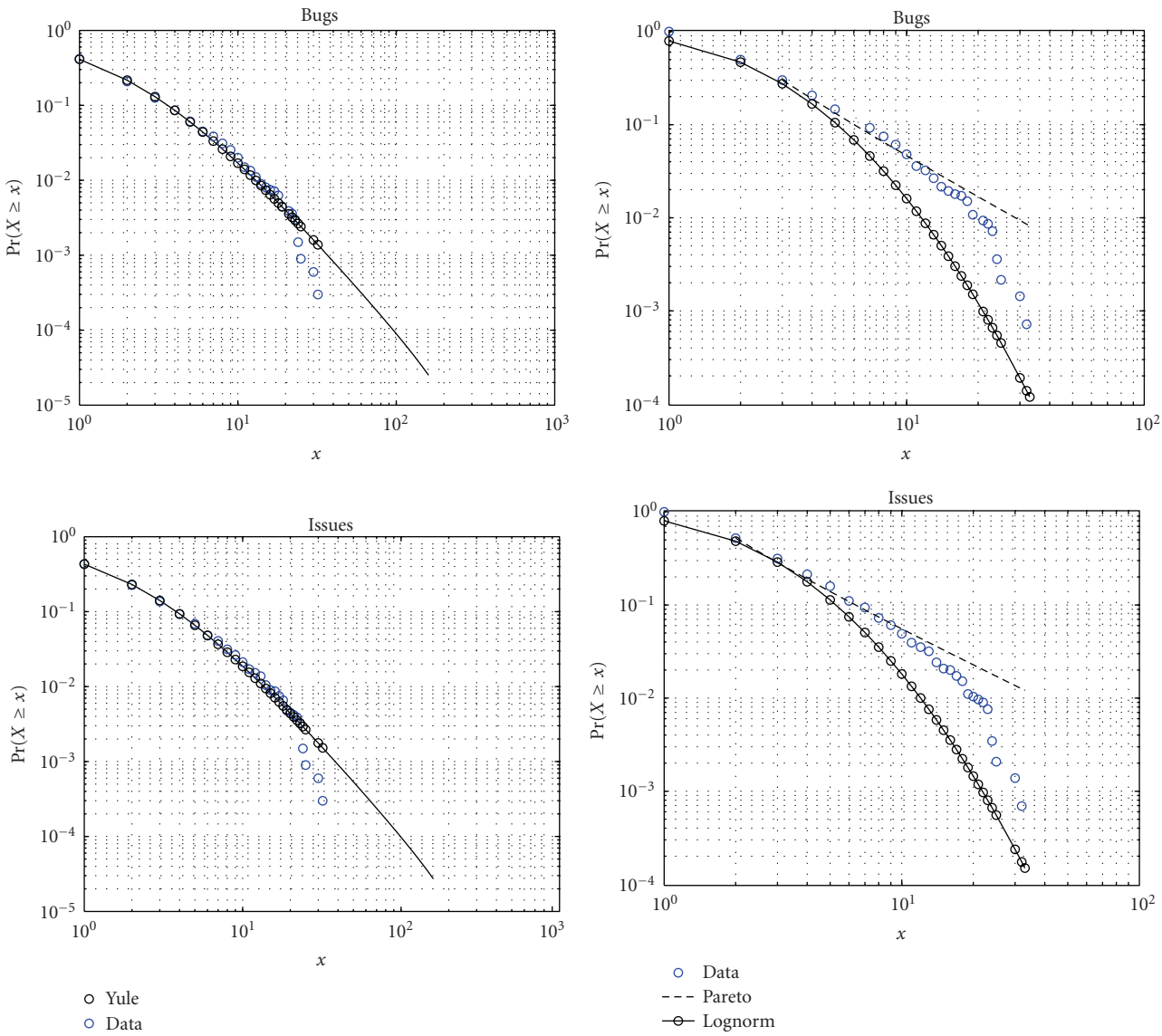


Figure 7: Empirical CCDFs of Bugs and Issues in Netbeans 6.0, with their best-fit theoretical distributions. Yule-Simon fit is shown separately.

will refer only to Spearman correlation. Using the latter, data must be ranked, with the correlation coefficient being given by

$$\rho_{\mathrm{SP}} = 1 - \frac{6\sum_i d_i^2}{n(n^2-1)}, \tag{5}$$

where $d_i$ are the differences among the ranks of each observation.

We report the correlations only for Eclipse-2.1 and for Netbeans-3.2, as representative of all the other releases. Tables 5 and 6 report correlation data for Eclipse-2.1, using Pearson and Spearman coefficients, respectively. Table 7 reports Spearman coefficients for Netbeans-3.2. The correlation coefficients in all other releases of the same system are substantially similar to those reported here, for both Eclipse and Netbeans.

The higher correlations are among Issues and Bugs, as it is natural, being one a subset of the other. This means that nodes having a high number of Issues also tend to have a high number of Bugs. In other words, the number of Bugs is always about the same fraction of Issues. Thus only one of them will be included in the subsequent analysis.

We computed the correlation matrix among Issue, Bug, CK metrics, LOC, Fan-out, Fan-in, and EGO-metrics. Correlations are almost the same in each release, with fluctuations generally below 10%.

In Eclipse, CK metrics, LOCS, Fan-Out, and EGO metrics generally show a moderate correlation with respect to Issues (Bug). In Netbeans, we have similar correlations, though usually slightly smaller. In both cases, the predictive power of these metrics is similar for the same software system. In both systems, LOC metric is the most correlated

Table 5: Eclipse 2.1. Pearson correlation among metrics.

| | Num. issue | Num. bug | LOCS | WMC | RFC | LCOM | CBO | Fan-in | Fan-out | Reach efficiency | Eff.size | Closeness | Dwreach | Infocen-trality | Size | Ties | Nweak-comp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Numbug | 97%** | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| LOCS | 53%** | 53%** | | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| WMC | 49%** | 48%** | 57%** | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| RFC | 58%** | 59%** | 68%** | 92%** | — | — | — | — | — | — | — | — | — | — | — | — | — |
| LCOM | 32%** | 30%** | 19%** | 79%** | 62%** | — | — | — | — | — | — | — | — | — | — | — | — |
| CBO | 54%** | 55%** | 65%** | 41%** | 70%** | 11%** | — | — | — | — | — | — | — | — | — | — | — |
| Fanin | 18%** | 17%** | 62%** | 30%** | 26%** | 26%** | 4%** | — | — | — | — | — | — | — | — | — | — |
| Fanout | 51%** | 52%** | 62%** | 30%** | 58%** | 3%** | 94%** | −1% | — | — | — | — | — | — | — | — | — |
| Reachefficiency | 0% | 1% | 4%** | −4%** | −1% | −3%** | 9%** | −14%** | 14%** | — | — | — | — | — | — | — | — |
| Effsize | 30%** | 29%** | 25%** | 36%** | 40%** | 26%** | 29%** | 96%** | 26%** | −10%** | — | — | — | — | — | — | — |
| Closeness | −2% | −2% | −1% | −1%** | −2% | 0% | −3% | −1% | −3%* | −5%** | −2%* | — | — | — | — | — | — |
| Dwreach | 27%** | 27%** | 23%** | 17%** | 29%** | 4%** | 46%** | 19%** | 50%** | 44%** | 32%** | −18%** | — | — | — | — | — |
| Infocentrality | −2%* | −2%* | −2% | −2% | −3%* | 0% | −4%** | −1% | −5%** | −61%** | −2%* | 94%** | −25%** | — | — | — | — |
| Size | 32%** | 31%** | 28%** | 38%** | 42%** | 26%** | 32%** | 95%** | 29%** | −10%** | 100%** | −2% | 34%** | −3%* | — | — | — |
| Ties | 32%** | 31%** | 27%** | 43%** | 45%** | 37%** | 27%** | 87%** | 23%** | −9%** | 89%** | −1% | 21%** | −2% | 89%** | — | — |
| Nweakcomp | −23%** | −23%** | −27%** | −18%** | −28%** | −2%* | −39%** | −14%** | −40%** | −2%** | −21%** | 4%** | −15%** | 5%** | −25%** | −22%** | — |
| Brokerage | 16%** | 15%** | 9%** | 30%** | 26%** | 35%** | 8%** | 85%** | 3%* | −5%** | 83%** | 0% | 12%** | −1% | 82%** | 88%** | −7%** |

** Correlation is significant at the 0.01 level. * Correlation is significant at the 0.05 level.

Table 6: Eclipse 2.1. Spearman correlation among metrics.

| | Num. issue | Num. bug | LOCS | WMC | RFC | LCOM | CBO | Fan-in | Fan-out | Reach efficiency | Effsize | Closeness | Dwreach | Infocen-trality | Size | Ties | Nweak-comp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Numbug | 95%** | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| LOCS | 46%** | 46%** | | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| WMC | 38%** | 38%** | 84%** | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| RFC | 46%** | 46%** | 90%** | 89%** | — | — | — | — | — | — | — | — | — | — | — | — | — |
| LCOM | 34%** | 34%** | 66%** | 85%** | 74%** | — | — | — | — | — | — | — | — | — | — | — | — |
| CBO | 45%** | 45%** | 78%** | 61%** | 86%** | 50%** | — | — | — | — | — | — | — | — | — | — | — |
| Fanin | 8%** | 7%** | 7%** | 26%** | 7%** | 26%** | −14%** | — | — | — | — | — | — | — | — | — | — |
| Fanout | 44%** | 44%** | 77%** | 60%** | 84%** | 51%** | 95%** | −15%** | — | — | — | — | — | — | — | — | — |
| Reachefficiency | 16%** | 16%** | 29%** | 17%** | 35%** | 13%** | 48%** | −29%** | 52%** | — | — | — | — | — | — | — | — |
| Effsize | 41%** | 41%** | 59%** | 59%** | 68%** | 56%** | 63%** | 45%** | 66%** | 21%** | — | — | — | — | — | — | — |
| Closeness | 38%** | 39%** | 51%** | 45%** | 59%** | 42%** | 62%** | 14%** | 66%** | 63%** | 72% | — | — | — | — | — | — |
| Dwreach | 40%** | 40%** | 54%** | 48%** | 62%** | 45%** | 66%** | 15%** | 70%** | 68%** | 76%** | 96%** | — | — | — | — | — |
| Infocentrality | −33%** | −34%** | −45%** | −35%** | −48%** | −35%** | −53%** | −2%* | −55%** | −51%** | −59%* | −79%** | −83%** | — | — | — | — |
| Size | 43%** | 42%** | 63%** | 62%** | 71%** | 58%** | 66%** | 46%** | 69%** | 22%** | 98%** | 72%** | 76%** | −59%** | — | — | — |
| Ties | 42%** | 42%** | 64%** | 60%** | 69%** | 56%** | 65%** | 41%** | 67%** | 17%** | 17%** | 65%** | 69%** | −65%** | 94%** | — | — |
| Nweakcomp | −28%** | −28%** | −48%** | −41%** | −47%** | −37%** | −44%** | −23%** | −44%** | −3%** | −42%** | −31%** | −34%** | 51%** | −52%** | −71%** | — |
| Brokerage | 42%** | 42%** | 61%** | 60%** | 69%** | 57%** | 65%** | 45%** | 67%** | 21%** | 100%** | 73%** | 76%** | −59%** | 99%** | 91%** | — |

** Correlation is significant at the 0.01 level. * Correlation is significant at the 0.05 level.

Table 7: Netbeans 3.2. Spearman correlation among metrics.

| | Num. issue | Num. bug | LOCS | WMC | RFC | LCOM | CBO | Fan-in | Fan-out | Reach efficiency | Effsize | Closeness | Dwreach | Infocen-trality | Size | Ties | Nweak-comp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Numbug | 98%** | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| LOCS | 47%** | 46%** | %** | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| WMC | 44%** | 42%** | 87%** | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| RFC | 44%** | 42%** | 87%** | 95%** | — | — | — | — | — | — | — | — | — | — | — | — | — |
| LCOM | 41%** | 39%** | 70%** | 87%** | 81%** | — | — | — | — | — | — | — | — | — | — | — | — |
| CBO | 33%** | 32%** | 58%** | 53%** | 71%** | 43%** | — | — | — | — | — | — | — | — | — | — | — |
| Fanin | 13%** | 12%** | 19%** | 34%** | 30%** | 31%** | 13%** | — | — | — | — | — | — | — | — | — | — |
| Fanout | 45%** | 44%** | 65%** | 58%** | 68%** | 54%** | 71%** | 3%** | — | — | — | — | — | — | — | — | — |
| Reachefficiency | 17%** | 16%** | 18%** | 14%** | 18%** | 15%** | 18%** | −19%** | 49%** | — | — | — | — | — | — | — | — |
| Effsize | 38%** | 36%** | 52%** | 58%** | 62%** | 54%** | 53%** | 56%** | 70%** | 20%** | — | — | — | — | — | — | — |
| Closeness | 38%** | 36%** | 43%** | 43%** | 46%** | 43%** | 38%** | 14%** | 70%** | 73%** | 61%** | — | — | — | — | — | — |
| Dwreach | 37%** | 35%** | 45%** | 45%** | 48%** | 44%** | 40%** | 17%** | 73%** | 77%** | 66%** | 94%** | — | — | — | — | — |
| Infocentrality | −26%** | −25%** | −30%** | −23%** | −23%** | −24%** | −14%** | 9%** | −38%** | −37%** | −26%** | −54%** | −60%** | — | — | — | — |
| Size | 39%** | 38%** | 55%** | 61%** | 65%** | 57%** | 56%** | 57%** | 73%** | 23%** | 98%** | 64%** | 69%** | −28%** | — | — | — |
| Ties | 39%** | 37%** | 54%** | 57%** | 60%** | 53%** | 50%** | 48%** | 65%** | 10%** | 83%** | 52%** | 57%** | −47%** | 88%** | — | — |
| Nweakcomp | −26%** | −25%** | −37%** | −32%** | −32%** | −31%** | −23%** | −16%** | −30%** | 12%** | −25%** | −17%** | −20%** | 60%** | −36%** | −63%** | — |
| Brokerage | 38%** | 37%** | 53%** | 59%** | 63%** | 55%** | 54%** | 56%** | 71%** | 21%** | 10%** | 62%** | 67%** | −28%** | 99%** | 84%** | −30%** |

** Correlation is significant at the 0.01 level. * Correlation is significant at the 0.05 level.

with Issues. This is expected, because bigger files have a larger chance to produce Issues and Bugs. However, other good predictors of Issues—comparable with LOC—are RFC, Fan-out, Size and, to a lesser extent, LCOM, Ties and Brokerage. In general, we observe that many SNA metrics are quite correlated with the number of Issues (and Bugs), showing the importance of considering these metrics.

In both Eclipse and Netbeans, Fan-in always shows a small—though significant—correlations with Issues. The different correlation between Fan-in and Fan-out with respect to Issues, indicates that to identify a fault-prone node it is important to take into account not only the number of links but also their direction. An Out-link directed from a compilation unit A to a compilation unit B may be considered like a channel easing the propagation of defects from B to A, but not vice versa. This fact highlights the importance of an analysis of a software system as an oriented graph.

CK and LOC metrics correlations with Issues are in line with results previously shown in [5]. In Eclipse, correlations between CK metrics and Eff-Size, Closeness, Size, Ties, brokerage are quite large. Correlations with Nweak-comp, Infocentrality, Dwreach, and Closeness are smaller. Only a minor correlation exists between CK metrics and Reach-Efficiency.

In Netbeans, correlations between CK metrics and Eff-Size, Size, Ties, Brokerage are also large. Smaller correlations hold between CK metrics and Closeness, Nweakcomp, Dwreach. Only minor correlations, like in Eclipse, exist between CK metrics, Reach-Efficiency, and Info-Centrality.

In both Eclipse and Netbeans, the only metrics that are anticorrelated with the number of Issues are Info-Centrality and Nweak-Comp, suggesting that it is better for a CU to have a high Information Centrality and Normalized number of Weak Components, to be less prone to get Issues and Bugs.

Most Eclipse and Netbeans EGO metrics are not strongly correlated with each other. For example, Reach-Efficiency has small correlation with Eff-Size, Size, and Brokerage, and no correlation with Nweakcomp. Size metric is the most correlated with the others EGO-metrics and shows an almost perfect correlation with Eff-Size and Brokerage. Consequently, it is clearly needed to consider just one of these metrics. We suggest to use Size, which is easier to compute and, at least in the considered systems, looks slightly better correlated to Issues.

These findings related to correlations answer our last two research questions.

**R4**: *are there SNA metrics significantly correlated with software Bugs, and to which extent?*

The data reported, and data very similar to them related to all other considered releases of Eclipse and Netbeans, confirm that there are significant correlations between several SNA metrics and the number of Bugs. These correlations are of the same order of magnitude of more traditional CK metrics—whose predictive power in predicting faulty classes has been studied and assessed for a long time [5, 7]. Note that all CK metrics, and most SNA metrics, are basically complexity metrics, denoting high coupling and/or low cohesion of the measured module. This is consistent with the positive correlation between these metrics and the fault-proneness of the module. However, some SNA metrics are anticorrelated to a fairly high extent with the number of Bugs, and this property might be further studied and exploited.

**R5**: *are there SNA metrics significantly correlated to traditional CK metrics, and to which extent?*

The study of Tables 6 and 7 confirms that all SNA metrics are significantly correlated to all the four considered CK metrics—WMC, RFC, CBO, and LCOM. Some SNA metrics— namely, Eff-Size, Size, Ties, and Brokerage—show quite high Spearman correlation coefficients with all these CK metrics.

## 9. Providing Estimates

In this section, we discuss how it is possible to estimate some values for the metrics starting from the knowledge of the analytical fitting functions. We assume that all the data are known for one system release and assume the persistence of the distributions across releases.

Let us consider, for instance, the metric Ties, and the Eclipse releases from 2.1 to 3.3. Let us start with the lognormal distribution. If we compute the estimate of the mean values using the best fitting parameters found, using the usual formula ($\exp(\mu + \sigma^2/2)$), they match actual values with an error of about 15% (see Table 9). With regard to the standard deviation, however, the estimate of the lognormal fails. In fact, empirical data show a systematic increase of their standard deviation, while the lognormal provides a constant value, since the best fitting parameters are almost constant.

It is also possible to estimate the expected maximum value for a lognormal population of finite size $n$, which depends on $n$, using the formula [37]:

$$\log(x_{\max}) = \mu + \sigma\sqrt{2\log(n)} - \sigma\frac{(\log\log(n) + \log(4\pi))}{2\sqrt{2\log(n)}} + \epsilon, \tag{6}$$

where $\epsilon$ is a small error term. We approximated our estimates using the first two terms, since the third is negligible in our case. The predicted extreme values for the Ties distribution are reported in Table 9, which shows a discrepancy with the empirical values of about 15/20%, which increases with the system size.

If we consider the best-fit power-law distribution, its exponent $\alpha_{\mathrm{PL}}$ has always values between 2 and 3, and this is consistent with the power-law property that, for such values of $\alpha$, the mean is finite, while the standard deviation diverges. In the case of a finite number of samples, this means that the standard deviation has obviously a finite value, but it tends to increase with the number of samples [15]. This is exactly the behavior which we observed. Therefore, when the number of CU increases from a release to another, so does the standard deviation. Note that the power-law cannot fit the bulk of the data, since the cut-off starts at about 140. So, it cannot be used to estimate the mean of the samples.

Table 8: The best fitting parameters for the three different distributions for the metric Ties. For each version of Eclipse, empirical first and second moment, number of CU and maximum value are also reported.

| Ties | | lognormal | | Power-law | | Yule-Simon | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Release | $\mu$ | $\sigma_{LN}$ | $\alpha_{PL}$ | $x_0$ | $\alpha_{YS}$ | $c$ | #CU | $\langle x \rangle$ | $\langle x^2 \rangle$ | |
| 2.1 | 2.85 | 1.54 | 2.38 | 174 | 2.23 | 20.6 | 7545 | 59.6 | 227.3 | 9799 |
| 3.0 | 2.80 | 1.54 | 2.39 | 141 | 2.21 | 19.1 | 10288 | 59.2 | 257.6 | 11901 |
| 3.1 | 2.85 | 1.56 | 2.37 | 143 | 2.16 | 18.6 | 11854 | 64.9 | 294.7 | 14711 |
| 3.2 | 2.82 | 1.57 | 2.35 | 141 | 2.14 | 17.7 | 14138 | 65.3 | 316.2 | 17029 |
| 3.3 | 2.83 | 1.57 | 2.33 | 145 | 2.13 | 17.4 | 15439 | 66.8 | 336.3 | 18819 |

Table 9: Estimates for the extreme values of the metric Ties. In the last column, the two values refer to the estimate obtained using parameters from release 2.1, or using parameters from the immediate previous version, respectively.

| Release | Actual value | $\langle x_{\max} \rangle_{\text{lognormal}}$ | $\langle x_{\max} \rangle$ (Eq. (8), $\alpha_{\text{power-law}}$) | $\langle x_{\max} \rangle$ (Eq. (8), $\alpha_{YS}$) |
|---|---|---|---|---|
| 3.0 | 11901 | 12962 | 12268 | 12609/== |
| 3.1 | 14711 | 13634 | 13594 | 14148/14234 |
| 3.2 | 17029 | 14511 | 15446 | 16327/16838 |
| 3.3 | 18819 | 14967 | 16463 | 17539/18363 |

Using the power-law, however, we may provide an estimate for the maximum value, a quantity more relevant than the estimate of the mean. It is well known that the following formula holds [15]:

$$\langle x_{\max} \rangle \sim n^{1/(\alpha-1)}. \tag{7}$$

So, for two generic releases we can write

$$\frac{\langle x_{\max_1} \rangle}{\langle x_{\max_2} \rangle} = \left(\frac{n_1}{n_2}\right)^{1/(\alpha-1)}, \tag{8}$$

and we can use one extreme value measured from release $i$ to estimate the extreme value of release $i+1$, when CU numbers are known. Using the values in Table 8, the error is about 15%, as reported in Table 9.

The Yule-Simon distribution is a good compromise between the two other considered distributions, because it fits both the bulk and the tail of the data. We numerically estimated the average using the best fitting parameters of the Yule-Simon distribution in Table 8, and they are in agreement with the empirical values. The power-law exponent obtained from the Yule-Simon best fit is among two and three, and it is consistent with the empirical standard deviation, which seems to diverge with the number of CUs. Furthermore, since (7) holds asymptotically, we can use the power-law exponent as obtained from the Yule-Simon best fitting in (8), to estimate the extreme values as before. These are in excellent agreement with the empirical results (Table 9).

We may now answer to the third research question **R3**: *is it possible to use these distributions to estimate the metrics values in subsequent releases?*

We found that mean values, as obtained from the analytical distributions, are in agreement with the empirical ones. From the knowledge of the best fitting parameters of the Yule-Simon distribution in one release, assuming persistence, we estimated the extreme values of subsequent releases using the CU number. Such estimates are in agreement with the empirical values with an error of $\Delta x/x = 456/18819 \simeq 2.5\%$.

These results have been obtained for the metric Ties for Eclipse but similar considerations hold also for the other metrics which are best fitted using Yule-Simon distribution.

## 10. Conclusions

In this paper, we studied for the first time the distribution of SNA metrics in OO software networks, comparing their properties with those of CK metrics and other graph-related metrics. We used as a central concept the Compilation Unit and not the class, to be able to better study the impact of metrics on Bugs and Issues, which always refer to CUs and not to classes, in commonly used configuration management systems.

The empirical distributions of all the studied metrics systematically present power-laws in their tails. This property holds also for bug distribution. It must be noted that bug distributions may be biased due to the lack of information in CVS commits, thus our results on bug distributions are as reliable as the information about bugs extracted from CVSs. All metrics have very similar features and shapes across all the system releases and also show very similar behavior in both Eclipse and Netbeans systems.

We found analytical distribution functions suitable for fitting the empirical data. Power-law always outperforms other fittings in the tails, whereas Yule-Simon distribution follows the shapes of most metrics empirical distributions very well. In particular, Ties and Fan-in metrics are fitted by Yule-Simon distribution from the very beginning of values, with the determination coefficients being over 0.98. We have shown—using the metric Ties—how it is possible to provide reliable estimates for averages and extreme values of subsequent releases from the knowledge of the best fitting

parameters and system size. The knowledge of extreme values of metrics could be exploited to keep under control the quality of software systems, because in general high values of these metrics denote high coupling among classes.

Regarding correlations among SNA metrics and Bugs, they are generally good, and when using the Spearman coefficient to assess them, they are comparable to those of CK metrics. It is known that LOC is one of the metrics best correlated with the number of defects. Nevertheless, as it holds for some other complexity metrics, they focus only on single software elements, while the use of SNA metrics allows to take into account the role of interactions between elements, and how these interactions correlate with defects. Consequently, we can state that the new SNA metrics are worth studying in greater detail, to better assess their predictive power regarding Issues and Bugs, maybe in conjunction, and not as an alternative to more traditional OO metrics.

Future developments of this seminal work will include controlled experiments to better understand the effect of SNA metrics on bug proneness and if they are able to identify different kind of bugs, and the construction of software graphs where the link direction and type are taken into account.

## References


[1] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[2] F. Brito e Abreu, "The MOOD metrics set," in *Proceedings of ECOOP Workshop on Metrics*, 1995.

[3] M. Lorenz and I. Kidd, *Object Oriented Software Metrics: A Pratical Guide*, Pretience Hall, Englewood Cliffs, NJ, USA, 1994.

[4] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial use of metrics for object-oriented software: an exploratory analysis," *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 629–639, 1998.

[5] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[6] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[7] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[8] S. Valverde, R. Ferrer Cancho, and R. V. Solé, "Scale-free networks from optimal design," *Europhysics Letters*, vol. 60, no. 4, pp. 512–517, 2002.

[9] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 687–708, 2007.

[10] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free geometry in object-oriented programming," *Communications of the ACM*, vol. 48, no. 5, pp. 99–103, 2005.

[11] J. P. Scott, "Social network analysis," *Sociology*, vol. 22, no. 1, pp. 109–127, 1988.

[12] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 531–540, May 2008.

[13] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, article no. 2, 2008.

[14] A. Tosun, B. Turhan, and A. Bener, "Validation of network measures as indicators of defective modules in software systems," in *Proceedings of the 1st International Conference on Predictor Models (PROMISE '09)*, 2009.

[15] M. E. J. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, no. 5, pp. 323–351, 2005.

[16] "Eclipse," http://www.eclipse.org/.

[17] "NetBeans," http://www.netbeans.org/.

[18] "Bugzilla," http://www.bugzilla.org/.

[19] http://netbeans.org/bugzilla/report.cgi.

[20] Cvs, http://www.nongnu.org/cvs/.

[21] H. Simon, "On a class of skew distribution functions," *Biometrika*, vol. 42, pp. 425–440, 1955.

[22] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "On the suitability of Yule process to stochastically model some properties of object-oriented systems," *Physica A*, vol. 370, no. 2, pp. 817–831, 2006.

[23] T. J. McCabe, "Complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[24] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Proceedings of the 6th Annual Conference Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, vol. 26, pp. 197–211, 1991.

[25] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 273–286, 2007.

[26] H. Zhang, "On the distribution of software faults," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 301–302, 2008.

[27] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative analysis of evolving software systems using the gini coefficient," in *Proceedings of IEEE International Conference on Software Maintenance (ICSM '09)*, pp. 179–188, September 2009.

[28] S. Valverde and R. V. Solé, "Network motifs in computational graphs: a case study in software architecture," *Physical Review E*, vol. 72, no. 2, Article ID 026107, pp. 1–8, 2005.

[29] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *Proceedings of the 3rd IEEE International Workshop onSource Code Analysis and Manipulation (SCAM '03)*, 2003.

[30] G. Baxter, M. Frean, J. Noble et al., "Understanding the shapeof Java software," in *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '06)*, Portland, Ore, USA, October 2006.

[31] G. Concas, M. Marchesi, A. Murgia, S. Pinna, and R. Tonelli, "Assessing traditional and new metrics for object-oriented systems," in *Proceedings of the Workshop on Emerging Trends in Software Metrics (ICSE '10)*, Cape Town, South Africa, May 2010.

[32] M. Eaddy, T. Zimmermann, K. D. Sherwood et al., "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[33] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta, "Threats on building models from cvs and bugzilla repositories:the mozilla case study," in *Proceedings of the 17th Annual International Conference on Computer Science and Software Engineering (CASCON '07)*, Toronto, Canada, October 2007.

[34] G. Yule, "A mathematical theory of evolution based on the conclusions of dr. j.c. willis," *Philosophical Transactions of the Royal Society London B*, vol. 213, pp. 21–87, 1925.

[35] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Review*, vol. 51, no. 4, pp. 661–703, 2009.

[36] Qualitas Research Group, "Qualitas Corpus Version 20090202," The Universityof Auckland, February 2009, http://www.cs.auckland.ac.nz/ewan/corpus.

[37] N. D. Singpurwalla, "Extreme values from a lognormal law with applications to air pollution problems," *Technometrics*, vol. 14, no. 3, pp. 703–711, 1972.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 820103, 12 pages
doi:10.1155/2010/820103

*Research Article*

# Exploring the Eradication of Code Smells: An Empirical and Theoretical Perspective

**S. Counsell,[1] R. M. Hierons,[1] H. Hamza,[1] S. Black,[2] and M. Durrand[3]**

[1] *Department of IS and Computing, Brunel University, Uxbridge UB8 3PH, UK*
[2] *Department of Information and Software Systems, University of Westminster, Harrow Campus, London HA1 4TP, UK*
[3] *change-s.com, Westminster Borough, London, UK*

Correspondence should be addressed to S. Counsell, steve.counsell@brunel.ac.uk

Received 2 September 2010; Revised 31 December 2010; Accepted 31 December 2010

Academic Editor: Giulio Concas



Code smells reflect code decay, and, as such, developers should seek to eradicate such smells through application of "deodorant" in the form of one or more refactorings. However, a relative lack of studies exploring code smells either theoretically or empirically when compared with literature on refactoring suggests that there are reasons why smell eradication is neither being applied in anger, nor the subject of significant research. In this paper, we present three studies as supporting evidence for this stance. The first is an analysis of a set of five, open-source Java systems in which we show very little tendency for smells to be eradicated by developers; the second is an empirical study of a subsystem of a proprietary, C# web-based application where practical problems arise in smell identification and the third, a theoretical enumeration of smell-related refactorings to suggest why smells may be left alone from an effort perspective. Key findings of the study were that first, smells requiring application of simple refactorings were eradicated in favour of smells requiring more complex refactorings; second, a wide range of conflicts and anomalies soon emerged when trying to identify smelly code; an interesting result with respect to comment lines was also observed. Finally, perceived (estimated) effort to eradicate a smell may be a key factor in explaining why smell eradication is avoided by developers. The study thus highlights the need for a clearer research strategy on the issue of code smells and all aspects of their identification and measurement.

## 1. Introduction

Bad code smells are a concept described in Fowler [1] to denote code that "screams out" to be refactored. In other words, it reflects code that is decaying and, unless eradicated, is likely to be the cause of frequent future maintenance, potential faults, and associated testing overheads. Left unchecked and free to fester over time, smells can soon become "stenches" with the potential for relatively high fault-proneness, added maintenance demands, and extratesting as outcomes. An evolving system left to decay is a problem "stored-up" and growing in size for a later date. Eradication of a smell is usually achieved through application of a single, or set of, refactoring/s, and twenty-two different code smells are described by Fowler in [1] together with the refactorings that are needed to remedy those smells. While the related topic of refactoring has been the subject of a significant number of research and other studies [2–9], the empirical and theoretical study of code smells seems to have been largely overlooked by the research community. Even the terminology used in code smell research has yet to find a firm footing and general acceptance. And yet, the problem of code smells has strong industrial resonance—decaying systems consume vast developer resources.

In this paper, we describe three studies of supporting evidence to justify our stance that a fresh look needs to be made of the area, the scope for research in the area, and the benefits that analysis of code smells could provide. The first presents a "smell-to-refactoring" theoretical justification for why some smells may be prohibitive for developers to remedy based on the number of related refactorings required to eradicate a smell; in other words, we suggest that the necessary effort required for smell eradication might itself be prohibiting factor. The second is a study of multiple versions

of five, Java open-source systems (OSSs) [2] from which extracted refactorings, when reverse engineered, showed little empirical propensity on the part of the OSS developers to eradicate smells. Finally, we describe an empirical examination of two versions of a proprietary C#, web-based system in which we point to why even identifying simple smells such as "large" classes and "long" methods [1] pose practical difficulties and raise both conflicts and anomalies. Results showed that perceived (estimated) effort to eradicate a smell may be a key factor in explaining why smell eradication is avoided by developers. Only limited evidence of smell eradication by developers and a wide range of practical problems soon emerged when trying to identify "smelly" code from proprietary C# code. The remainder of the paper is organized as follows. In Section 2, we present the motivation for the work. We then present each of the three studies in the order described with supporting data (Sections 3, 4, and 5, resp.) and describe threats to study validity in Section 6. We then finalize with conclusions and future work (Section 7).

## 2. Motivation/Related Work

The research in this paper is motivated by one overarching research question: why, if the eradication of code smells provides such obvious potential, theoretical benefits and is a problem that all code might suffer from as it evolves, has the same topic received such little research attention? This question itself induces a range of other questions and motivating factors. First, what role does human judgment and motivation for eradicating smells fulfill in the identification of smells? In other words, are developers interested in smell eradication? It is clear that the choice of what a "large" class or "excessive" coupling constitutes is subjective and this might lie at the heart of why developers are reluctant to address code smells. Second, what anomalies and inconsistencies arise when we attempt to "sniff out" smells from systems? That is, if we consciously search out code smells, what practical problems arise? Third, we need to consider the opportunity cost of choosing to eradicate one code smell over another. Developer time is limited, and there is a high opportunity cost of any smell eradication effort. Fourth, what theoretical considerations become important for the practical eradication of code smells? The activity of eradicating a single smell can, in theory, require a range of subactivities, depending on smell "complexity." Finally, we cannot discount from our discussion the burden that increased testing places on the developer. Just as when a developer undertakes a simple refactoring, for every smell eradicated there is a need to test the resulting code to ensure it has retained its "semantics"; eradication of a smell poses a similar challenge.

In terms of related work, research into code smells started promisingly with several industry-oriented studies, but seems to have petered out more recently. While smells are widely acknowledged as a problematic aspect of software development, very little research work has focused on code smells, their analysis and even less on the empirical study of smells from industrial, proprietary code. Two notable, seminal studies of code smells were undertaken by Mäntylä et al. [10] and Mäntylä and Lassenius [11, 12] who conducted an empirical study of industrial developers and their opinion of smells in evolving structures. The study gave insights into which smells developers most "understood" and hence they would be most likely to eradicate—the "Large Class" smell [1] featured prominently. A well-known "taxonomy" for allocating code smell was also proposed by Mäntylä in [13]; in subsequent work, Mäntylä and Lassenius also describe mechanisms for making refactoring decisions based on smell identification [11]. Recent research by Khomh et al. explored code smells using a Bayesian network approach [14] and from looking at changes made to a system as a basis for smell identification [15]. Counsell et al. established a link between refactoring and code smells in terms of the in- and out-degrees of a dependency diagram [2] supported with empirical OSS data.

Olbrich et al. [16] describe the study of two open-source systems over several years of development and focuse on two code smells in particular (the "God class" and "Shotgun Surgery" smells); different change behavior was observed for classes "infected" by code smells. More recently, Olbrich et al. [17] explore the change and fault proneness of "God" and "Brain" classes for systems ranging between seven and ten years old. Results showed that both smells were more fault and change prone, but when normalized for size were actually (and counter intuitively) less fault prone. Li and Shatnawi [18] investigated the link between bad smells and class error probability in an open-source system—some evidence to support high fault rates in smell code was reported. Van Emden and Moonen [19] investigated how the quality of code could be automatically assessed by checking for the presence of code smells and illustrated the feasibility of their approach through jCOSMO, a prototype code smell tool. A set of design flaws, including recognized code smells and a strategy, based on metrics for detecting those design flaws was described by Marinescu [20]. The mechanism was validated empirically. The same author refines that earlier work in [21]. Finally, Hamza et al. [22] provide an in-depth deconstruction of both Fowler's and Kerievsky's code smells [5] in an attempt to determine their overlap. While these studies have provided a basis for the area of code smells and the study of code smells, a range of open research issues persist. In the next three sections, we describe two empirical studies and one theoretical study (in that order) which question the viability of approaches to smell identification and eradication.

## 3. Refactorings Per Smell

*3.1. Data Analysis.* As a first part of our smell analysis, we describe the *potential* cost in time and effort of undertaking each of the 22 code smells. The basis of the analysis is that in Sections 4 and 5 we will see evidence of how limited smell eradication appears to be (Section 4) and some of the difficulties which arise when we attempt to sniff out code smells (Section 5). In this section, we provide a concrete suggestion as to why this might be the case. Earlier, we

stated how each of the code smells proposed by Fowler [1] could be eradicated by application of one or more other refactorings. Most refactorings (as well as having its own steps to completion) require other related refactorings to be undertaken in a nested relationship. Put another way, refactoring X might require refactoring Y, which in turn might require refactoring Z. Each of X, Y, and Z may also have other nested refactorings. All of X, Y, and Z can be extracted from Fowler's text as *remedies* for each of the smells. We can then posit that a factor inhibiting a developer addressing a code smell is the *total* number of refactorings that might need to be undertaken after following the "chains, induced by each of X, Y, and Z and used to remedy that smell. As part of our analysis, we therefore enumerated the refactorings that each of the smells induced, and this was achieved using a bespoke tool. Table 1 gives the 22 code smells listed in Fowler [1].

Figure 1 shows the potential number of refactorings that each of the 22 code smells requires. It is interesting that the smells observed in Section 4 are smells with relatively higher numbers of associated refactorings. The Large Class smell (number 10) has 40 associated refactorings. One of the reasons why this smell requires so many refactorings is due to requirement for the movement of methods to new class/classes and associated dependencies which, as we stated earlier, destroys class cohesiveness and forces the unpicking of all dependencies between methods. The Long Method smell (number 12) has 20 associated refactorings and the Lazy Class (number 11) 15 associated refactorings. As interestingly, these were the smells that we found difficult to tangibly identify from the ITWeb subsystem. On the other hand, the smells that we identified to be eradicated from the five OSS have relatively fewer required refactorings. Smell 1 actually has only 2 associated refactorings, and smell 16 has only 4 associated refactorings. Smells 7, 8, and 19 and 20 have relatively more associated refactorings overall but then again, we have no firm evidence that these were actually eradicated. Finally, the switch statement (smell 21) identified in one ITWeb class requires 16 separate refactorings in order to be eliminated—a relatively difficult smell to eradicate.

*It would seem that developers might well eradicate smells, but they tend to be smells that require little effort when compared with others.* Interestingly, and on a final note, in the developer survey carried out by Mäntylä [13], the Long Method smell stood out as the smell many developers "understood" the workings of most. In other words, conceptually speaking, developers know exactly what this smell arises from, the problems that it might pose and, more than likely, the means of eradicating this smell. One would therefore think that a greater understanding of a smell would imply that developers would naturally be more likely to want to eradicate that smell. However, we see that from Figure 1, the Long Method smell (smell 12) requires a relatively high amount of effort for its eradication. We therefore suggest that the effort required for the Long Method smell is a prohibiting factor for developers who might consider eradicating these smells. The lesson is that just because a smell is easy to understand does not mean it is easy to eradicate.

Table 1: Code smells and their description.

| Smell | Description |
|---|---|
| (1) Alternative classes with different interfaces | Two classes appear different on the outside, but are similar on the inside |
| (2) Comments | Comments should describe why the code is there not what it does |
| (3) Data class | Classes should not contain just data, they should contain methods as well |
| (4) Data clumps | Data that belong together should be amalgamated rather than remain separated |
| (5) Divergent change | Changes to code should be kept local; too many diverse changes indicate poor structure |
| (6) Duplicated code | Eradicate duplicated code whenever possible |
| (7) Feature Envy | Class features that use other class" features should be moved those other classes |
| (8) Inappropriate intimacy | Classes should not associate with other classes excessively |
| (9) Incomplete library class | Avoid adding a method you need (and which does not exist in a library class) to a random class |
| (10) Large class | A class has too many methods |
| (11) Lazy class | A class is doing too little to justify its existence |
| (12) Long method | A method is too large; it should be decomposed |
| (13) Long parameter list | A method has too many parameters |
| (14) Message chains | Avoid long chains of message calls |
| (15) Middle man | If a class is delegating too much responsibility, should it exist? |
| (16) Parallel inheritance hierarchies | When you make a subclass of one class, you need to make a subclass of another |
| (17) Primitive obsession | Overuse of primitive types in a class |
| (18) Refused bequest | If inherited behavior is not being used, is inheritance necessary? |
| (19) Shotgun surgery | Avoid cascading changes; limit the number of classes that need to be changed |
| (20) Speculative generality | Code should not be added for "just in case" scenarios—it should solve current problems |
| (21) Switch statements | Polymorphism should be used instead of large switch statements |
| (22) Temporary field | Classes should not contain unnecessary fields |

## 4. Open-Source Systems

*4.1. Data Analysis.* As a second part of our smell analysis, we use data from five, open-source Java systems as an empirical basis. These systems have been the basis of other empirical studies [2, 3, 23]. The criteria for initially choosing those

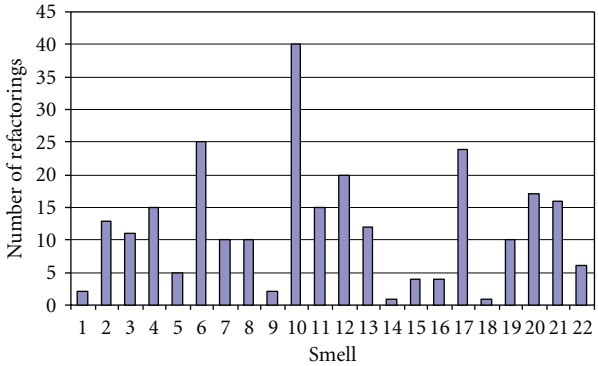


Figure 1: Code smells and their refactorings.

systems reported at length in those studies were that they had to be entirely Java systems and to have evolved over a minimum number of versions. From the same five systems we extracted a set of fourteen specific refactorings. The five systems studied were as follows.

(1) Antlr a framework for constructing compilers and translators using a source input of Java, C++ or C#. Antlr began with 153 classes and 31 interfaces. The latest version had 171 classes and 31 interfaces (five versions were studied).

(2) PDFBox a Java PDF library allowing access to components found in a PDF document. The initial system had 135 classes and 10 interfaces; the latest version of six had 294 classes and 52 interfaces.

(3) Velocity a template engine allowing web designers to access methods defined in Java. Velocity began with 224 classes and 44 interfaces. At the ninth version, it had 300 classes and 80 interfaces (nine versions were studied).

(4) Tyrant a graphical-based, fantasy adventure game, incorporates landscapes, dungeons and towns. The system began with 112 classes and 5 interfaces. At the ninth version, it had 138 classes and 6 interfaces (nine versions were studied).

(5) HSQLDB a relational database application supporting SQL. HSQLDB started with 52 classes and 1 interface. The latest version had 254 classes and 17 interfaces (four versions were studied).

The basis on which the initial study rests is that, to eradicate a smell, a specific set of refactorings need to be applied. Occurrences of fourteen specific refactorings were automatically extracted from multiple versions of these systems as part of an earlier study documented in [2]. The fourteen refactorings were chosen by two industrial developers as those most likely to be undertaken on a day-to-day basis and therefore ranged across OO concepts such as encapsulation and inheritance. Simpler refactorings for renaming/moving fields and methods were also included for the same reason. The refactorings were extracted by a bespoke tool and (in ascending order of frequency found in the five systems together with a brief description of each) are as follows.

(a) Encapsulate Downcast. "A method returns an object that needs to be downcast by its callers" [1]. (This was the least frequently applied refactoring.)

(b) Push Down Method. "Behavior on a superclass is relevant only for some of its subclasses" [1].

(c) Extract Subclass. A class has features used only in some instances—a subclass for that subset of features is created.

(d) Encapsulate Field. A field is made private.

(e) Hide Method. A method is made private.

(f) Pull Up Field. "Two subclasses have the same field. Move the field to the superclass" [1].

(g) Extract Superclass. Two classes have similar features. A superclass is created and common features moved.

(h) Remove Parameter. Parameter is unused by a method.

(i) Push Down Field. "A field is used only by some subclasses. Move the field to those subclasses" [1].

(j) Pull Up Method. Methods with identical results are moved to the superclass.

(k) Move Method. A method is moved to another class.

(l) Add Parameter. A parameter is added to a method.

(m) Move Field. A field is moved to another class.

(n) Rename Method. (This was the most frequently applied refactoring.)

*4.2. Research Question.* As part of the smell analysis, we first pose the question: *given the set of refactorings extracted from the five systems, which combination of those refactorings, applied to the versions of the five systems, have been used to remedy code smells*? In other words, from the data we collected on refactorings, do developers actually refactor (whether consciously or otherwise) to remedy smells and, if so, to what extent? The total set of 891 refactorings extracted by the tool over all versions of all systems was thus analyzed on a version-by-version basis to determine which smells they eradicated. The list of refactorings required for the reverse engineering process (i.e., to eradicate each smell) was provided in Fowler [1], and Table 1 gives the full list of 22 code smells. The process of deciding whether a smell had been entirely remedied required an exact match to be found between the list of refactorings specified by Fowler (to eradicate that smell) and a subset of refactorings extracted from the same version of a system. For a partial eradication, only a partial match between the extracted refactorings and that subset required to eradicate a smell was required. The smell analysis presented is based on *only* the refactorings extracted by the tool. The data on which refactorings had been extracted was analyzed using a spreadsheet, in which the frequency of each of the 15 refactorings was output on a version-by-version basis for each of the five systems. A sample of the set of 15 refactorings extracted was validated by the tool developers when the tool was run against the source code. This involved manual checking of the output

(i.e., the refactorings) against the Java source code. We thus have confidence in the correctness of the data and in the identification of the smells that we have identified as eradicated or partially eradicated.

Table 2 shows the five systems and the versions in each of the systems where some evidence of remedying of smells was found. For example, in versions 3 and 6 of the PDFBox system, a combination of refactorings was found to remedy smells 1 and 16. Column 3 of the table shows which smells were *entirely* remedied through application of refactorings. Column 4, on the other hand, shows the smells which *might* have been remedied.

Table 2 shows that we can only identify two smells as definitely having been remedied (i.e., smell 1 and 16). Smell 1 is "Alternative Classes with Different Interfaces." This smell occurs when two classes have a similar internal content but different external composition (i.e., in the parameter list). They should be amalgamated to present a common interface. Smell 16 is "Parallel Inheritance Hierarchies" where two separate inheritance hierarchies grow dependently and where creating subclasses in one requires subclasses to be created in the other. It is also worth noting that unexpectedly, later versions of the five systems did not show any greater propensity for smell eradication than earlier versions. This was surprising, since we might expect smells to worsen as a system evolved.

From Table 2, we see that following our analysis, only six of the remaining twenty smells *might* have been remedied according to column 4 (i.e., smells 3, 7, 8, 10, 19, and 20). Some of the fourteen refactorings identified from the systems have also been identified as "core" refactorings (i.e., are likely to be used frequently in multiple code modification scenarios). The "Move Method," "Move Field," and "Add Parameter" refactorings are typical examples [2]. These would be refactorings that we might expect a developer to apply as part of regular software maintenance and to be unconnected with conscious, intentional refactoring; it is difficult to know whether the developer actually set out to refactor or whether it was a byproduct of the day-to-day maintenance processes.

The results from Table 2 highlight the relative complexity of some smells over others, but the overriding message seems to be that only a small subset of smell eradications, from a small subset of the total number of versions from the five systems (13 versions from 33), were attempted. This claim has to be qualified with the caveat that we have no knowledge of whether any smell had been eradicated deliberately by the developer or whether the two refactorings were applied in combination at the same time to achieve the objective of smell eradication.

*4.2.1. Smell Decomposition.* Each of the six smells in Table 2 has a set of refactorings that need to be considered and then applied in order that they can be eradicated. For example, consider smell 1 "Alternative Classes with Different Interfaces." According to the Fowler mechanics, repeated application of the "Move Method" and "Rename Method" refactorings will remedy this smell. Equally, the "Parallel Inheritance Hierarchies" smell (smell 16) occurs when two separate inheritance hierarchies grow in a dependent fashion such that creating subclasses in one requires subclasses to be created in the other. Repeated application of the "Move Method" and "Move Field" refactorings would remedy this smell. It is the fact that these two smells require a relatively small number of frequently applied and overlapping refactorings (i.e., from the set of 14 extracted) that possibly account for the result in Table 2. All three refactorings (i.e., Move Method, Move Field, and Rename Method) appear in the top five refactorings from the list in Section 4.1. In other words, if the developer did set out to eradicate either of the two aforementioned smells, it may simply be because they only required the application of a set of two, relatively simple refactorings (and those that a developer regularly carried out) in each case.

Table 3 shows the total set of refactorings necessary to eradicate each of the six smells that were only partially remedied and illustrates that the frequently applied refactorings were, again, a common feature of the eradication process (i.e., Move Field, Move Method, and Rename Method appear often). The extent to which smells were only partially remedied is best illustrated with an example. From Table 3, code smell 10 (i.e., Large Class) requires the application of four refactorings in order to be eradicated. These four are "Extract Class," "Extract Subclass," "Extract Interface," and "Replace Data Value with Object". Only evidence of one of these four, namely, the "Extract Subclass" refactoring, was found from the extraction of refactorings by our tool. In other words, for this smell (and for many of the other five smells in Table 3) only a small minority of the *required* refactorings for eradication of smells were found to have been applied. The question arises as to why these smells were not totally eradicated? The "Large Class" smell (which occurs when a class is trying to do too much) requires the application of four refactorings; the first of these is "Extract Class" which decomposes a class into two or more separate classes. The "Extract Subclass" refactoring accounts

Table 2: Smells eradicated (and partially eradicated).

| System | Version | Smell remedied | "Partial" smells remedied |
|---|---|---|---|
| Antlr | 2 | 1, 16 | 7, 8, 19, 20 |
| PDFBox | 3 | 1, 16 | 7, 8, 19, 20 |
| | 6 | 1, 16 | 3, 7, 8,19, 20 |
| Velocity | 2 | 1, 16 | 3, 7, 8, 19 |
| | 3 | 1, 16 | 3, 7, 8, 19 |
| | 5 | 1, 16 | 3, 7, 8, 19 |
| | 6 | 1, 16 | 7, 8, 19 |
| | 9 | 1 | 20 |
| Tyrant | 7 | 1 | 20 |
| | 8 | 1 | 3, 20 |
| HSQLDB | 1 | 1, 16 | 7, 8, 19, 20 |
| | 2 | 1, 16 | 7, 8, 10, 19, 20 |
| | 3 | 1, 16 | 7, 8, 10, 19, 20 |

Table 3: Smells that were partially remedied.

| Code smell | Name | Refactorings |
|---|---|---|
| 3 | Data Class | Move Method<br>Encapsulate Field<br>Encapsulate Collection |
| 7 | Feature Envy | Move Method<br>Move Field<br>Extract Method |
| 8 | Inappropriate Intimacy | Move Method<br>Move Field<br>Change Bidirectional Association to Unidirectional<br>Replace Inheritance with Delegation<br>Hide Delegate |
| 10 | Large Class | Extract Class<br>Extract Subclass<br>Extract Interface<br>Replace Data Value with Object |
| 19 | Shotgun Surgery | Move Method<br>Move Field<br>Inline Class |
| 20 | Speculative Generality | Collapse Hierarchy<br>Inline Class<br>Remove Parameter<br>Rename Method |

for the case when the class being decomposed requires the creation of at least one subclass; the same rule applies to the "Extract Interface" refactoring in terms of decomposition of interfaces. The "Replace Data Value with Object" refactoring occurs when a data item needs additional data or behavior; the data item should be turned into an object. Each of these four refactorings required to eradicate the "Large Class" smell thus require significant effort to apply since they require either structural changes to the code or the introduction of objects to the system. While we did not collect all four refactorings, we suggest that the high-level structural nature of these refactorings was the reason why developers avoided their use in smell eradication.

The overriding message from the data in Tables 2 and 3 appears to be that while we can identify some evidence that refactorings associated with smells are undertaken, those refactorings appear to be ones that a developer might be liable to undertake anyway without any thought given to the eradication of any smell. For example, a developer might rename or move a method as part of general maintenance activity or in response to a fault fix. We are therefore skeptical about the claim that developers actively seek out code smells. The opposite may actually be the case; developers may actively avoid code smells because they are relatively difficult to tackle.

*4.2.2. The Remaining Smells.* Smell 3 "Data Class" is a class that has fields and getting and setting methods for those fields and are therefore merely data holders for other classes. This implies that they only exist to be manipulated by other classes. Smell 7 is "Feature Envy," which occurs when the methods of one class use the methods of another class excessively. Smell 8, "Inappropriate Intimacy," arises when two classes are coupled excessively to each other. The "Shotgun Surgery" smell arises when a change in one class requires cascading changes in several other classes. "Speculative Generality" arises when a developer adds functionality in the belief that it will be needed later on. We note that Mäntylä's taxonomy [13] rates this smell alongside duplicate code and dead code in terms of the harmful effect it might have on a system. While all of these smells require at least one simple refactoring, they also require the application of at least one "complex" refactoring. We suggest that the complexity of eradicating a smell (in some cases) is a factor in developers avoiding smell eradication.

## 5. A C# Proprietary System

*5.1. An Aggressively Refactored System.* The final part of our analysis is exploration of a C# subsystem for a web-based, loans system providing quotes and financial information for online customers; henceforward and for system anonymity we will refer to this system as simply "ITWeb." We examined two versions of one of its subsystems. The first, an early version, comprised 401 classes. A later version (henceforward version $n$) had been the subject of a significant refactoring effort to amalgamate, minimize as well as optimize classes—we were given no information as to which version it represented; it comprised 101 classes only and had thus been reduced in size by 300 classes through a process of aggressive refactoring (through merging of classes and optimization of others). For the purposes of this second analysis, we focused on four smells which, arguably, should be easily identifiable from the source code via simple metrics. These were as follows.

(1) Large Class. A class is trying to do too much, identified by a relatively large number of methods. Such a class is difficult to maintain and thus should be refactored—the class should be decomposed into two or more classes.

(2) Long Method. A method is doing too much, identified by its large number of executable statements. In the same way that as that for Large Class. The method should be decomposed into two or more methods.

(3) Lazy Class. A class is not doing enough to justify its existence, identified by a small number of methods and/or executable statements; it should be merged with its nearest, related class.

(4) Comments. There are conflicting opinions on the role that comments play in code from a smell perspective. Large numbers of comments in theory are useful, but, on the other hand might suggest that the relevant code is overly complex and thus needs significant explanation; an alternative viewpoint is that the code has been modified significantly and

the comments reflect the activity around a method or methods of a class over the course of its lifetime. Excessive comments should be treated with care—they may be a smell indicating problematic code.

The SourceMonitor tool [24] was used to extract a set of smell-relevant metrics [25] from each version.

*Metric 1* (Average Methods per Class). It is defined as the average methods per class for all class, interface, and struct methods. It is computed by dividing the total number of methods by the total number of classes, interfaces, and structs.

*Metric 2* (Average Statements per Method). The total number of statements found inside a class divided by the number of methods.

*Metric 3* (Average Calls per Method). The average number of calls to other methods inside all methods in a class (i.e., intracoupling). This metric does not include calls to methods of other classes in the same way that, for example, the Coupling between Objects metric of Chidamber and Kemerer does [26].

*Metric 4* (Average Class Complexity). It is defined as the sum of the complexities of the methods in a class divided by the number of methods and is in line with the definition provided by McConnell [27]. The complexity of a method is the count of the number of unique paths through a method. Each method therefore has a minimum complexity of "1" if it comprises just one path. A value of "1" is added to the value of the metric for each branch statement (if, else, for, while, etc.; a "1" is also added for each "||" or "&" operator in an "if" or "while" statement).

*Metric 5* (Percentage Comments). This metric reflects the percentage of lines of code accounted for by comments.

For each of these four metrics individual class values were also collected (i.e., the methods, statements, calls per method, complexity, and comments per class). We note that in several cases, a file can contain more than one class, in which case the average reported is that for the set of classes rather than the individual class.

*5.2. Research Question.* The research question on which we analyze ITWeb is as follows. *Can we expect the four aforementioned code smells to occur frequently in a system when we deliberately set out to "sniff" them*? Moreover, we might expect that since the system was aggressively refactored from version 1 to *n*, we might find less occurrences of the set of smells in the later version than in the earlier. Table 4 shows the summary data for versions 1 and *n* for the ITWeb system. We see that the average number of methods, average size (statements) of methods, average calls per method, average complexity decrease from version 1 to version *n* suggesting that the extensive refactoring that occurred between version 1 and *n* succeeded in reducing both class size and complexity. Percentage of comments also decreased from version 1 to version *n*, suggesting that developers deemed the removal of comments as a necessary aspect of the refactoring process. The question that arises is whether and/or to what extent either version presented opportunities for smell eradication? We explore each of the four listed smells in the order described to find out.

*5.2.1. Large Class.* To identify occurrences of the Large Class (LC) smell in version 1, one way of achieving this would be to order classes on descending Average Methods per Class and refine the search from there. By doing this, we find that the class with the largest number of methods is a *sealed* C# class (i.e., it cannot be inherited) called `PageController.cs`. This is an architectural, pattern-based class, essential for the coordination of ITWeb, and contains 80 methods. (Fowler defines the page controller pattern as "An object that handles a request for a specific page or action on a website".) Inspection of the code for this class revealed that each method handled one of a number of functionally cohesive requests for web page details. For example, there were methods to `SaveAccountDetails`, `SaveApplicationDetails`, and `SaveBrokerContactDetails`, and so forth. The average complexity of this class was 2.08, well below the average complexity for version 1. The number of calls per method was 4.5, well above the average. This last metric presents a conflict: a strong interdependence and coupling between the methods in this class is generally considered to contribute positively to the cohesiveness of the class, but equally would pose a huge problem if we wanted to decompose the class. The classes with the second and third largest numbers of methods are Controller and Navigation-based classes (`LoanSessionController.cs` and `PageNavigator.cs`, resp.), again with similar functionality to `PageController.cs`.

Ordering version *n* in the same way, we find that the maximum number of methods is 12 and belongs to a class called `ResultRowDTO.cs`. This is another architectural, pattern-based Data Transfer Object (DTO) [28]. A DTO wraps up data for transfer between two processes, possibly over a network, to prevent the overhead of multiple (remote) calls. Inspection of the code for this class confirmed that each of the 12 methods contained only a single "get" and "set" method. The average complexity of this class was 1, well below the average complexity for the subsystem of 1.13, and the number of calls per method was zero. In contrast to being classified as a smell, this class is a key element of the system architecture and has desirable properties only. The median of class size was 3 for this version, reinforcing the difficult question as to "which class to choose for eradication?" The problems that arise with the LC smell are therefore (a) deciding on what exactly constitutes a "large" class (a largely arbitrary choice) and (b) the fact that coupling between methods adds to class cohesion [29], yet makes an LC smell eradication more problematic due to dependencies. Figures 2(a) and 2(b) show the distribution of average methods per class in version 1 (Figure 2(a)) and version *n* (Figure 2(b)), respectively. The scale on the *y*-axis is an indication of the extent to which the average number of methods per class was reduced as a direct result of the refactoring effort. Only one

Table 4: Summary Data for ITWeb.

| Version | Classes | Ave. no. methods | Ave. statements/method | Ave. calls/method | Ave. complexity | %Comments |
|---|---|---|---|---|---|---|
| 1 | 401 | 5.59 | 4.03 | 2.24 | 1.70 | 4.0 |
| $N$ | 101 | 3.35 | 1.79 | 1.29 | 1.13 | 0.3 |

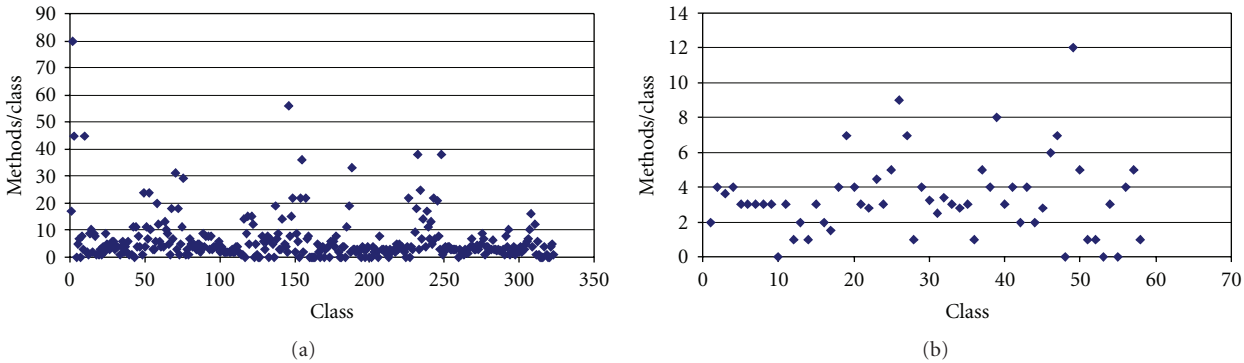


Figure 2: (a) Methods/class (version 1). (b) Methods/class (version $n$).

class in version $n$ has an average number of methods above 10. This compares with 45 values exceeding that value in version 1.

We conducted an 80/20 analysis of the values in each of Figures 2(a) and 2(b) to determine the distribution changes that had taken place from version 1 to version $n$ [30]. For Figure 2(a), 80% of values were found in approximately 136 (33%) of the 401 classes in total. For Figure 2(b), 80% of values were found in approximately 55 (approximately 54%) of the 101 classes, emphasizing the positive change in distribution of the values in the later version and a more even distribution as a result of the refactoring process. A short "head" and "long" tail are therefore less evident in the later version than in the former. The standard deviation of the values in Figure 2(a) is 8.61, compared with just 2.26 for Figure 2(b).

We note that the scale on the $x$-axis of Figures 2(a) and 2(b) represent files, within which more than one class may be housed (this explains the value of 350 and 70 on the respective axes and does not reflect the number of classes).

*5.2.2. Long Method.* If we now turn to the Long Method (LM) smell, one way of identifying such a smell is to order the set of classes in each version on Statements per Method and then refine the search from there. If we order version 1 in this way, we find that the class `ComparisonEngine.cs` contains the method with the highest number of statements. Inspection of the code revealed this method to contain one large *switch* statement comprising 340 statements. The switch statement is actually a smell itself (see Table 1, number 21), since OO suggests the use of the more appropriate OO facet of polymorphism instead of large decision control constructs [1]. However, we see no value in decomposing this method since for a web application switch statements effectively represent user interface choice (an essential aspect for guiding the online process).

Doing the same for version $n$, we found that the largest method was again a DTO class called `LoanDTO.cs` containing just one method. Inspection of the code revealed the method to be a set of similar, executable statements for returning a code to the main program depending on the values of user-filled fields (e.g., combinations of company name seeking online quotes, company logo, payment details, and payment rates). The average complexity of this class was 5, well above the average (1.13). However, the very nature of each method (comprising multiple "if" statements) does contribute to complexity as we have defined it. That said however, we see no obvious benefits to simplifying code complexity where the purpose of the code is obvious and essential to the workings of the system.

Three overriding messages from the LM analysis emerge. First, by necessity, some methods (often the longest) exist for a good reason. Second, searching out one smell can often lead to the identification of other, potentially more harmful smells. One aspect of the analysis presented is that by avoiding some smells, we may inadvertently miss others. Moreover, we have to consider the possibility that smells are created due to eradication of others; in the same vein, a study by Pietrzak and Walter used knowledge of already detected smells and the interrelationships between smells to detect further smell types [31]. Finally, necessary complexity and how that manifests in code is often infeasible to remove when existing code communicates the purpose of the code.

Figure 3(a) shows the distribution of statements per method for version 1 of the ITWeb system; Figure 3(b) shows the same distribution for version $n$ of the same system. As per the values in Figures 2(a) and 2(b), the contrast between the two set of values is marked in terms of the reduction shown in the second of the figures. The maximum value in Figure 3(b) is 8.33 compared with 38.63 in version 1 (Figure 3(a)). An 80/20 analysis of the values in each of Figures 3(a) and 3(b) revealed that for Figure 3(a), 80% of values were found in approximately 113 (approximately

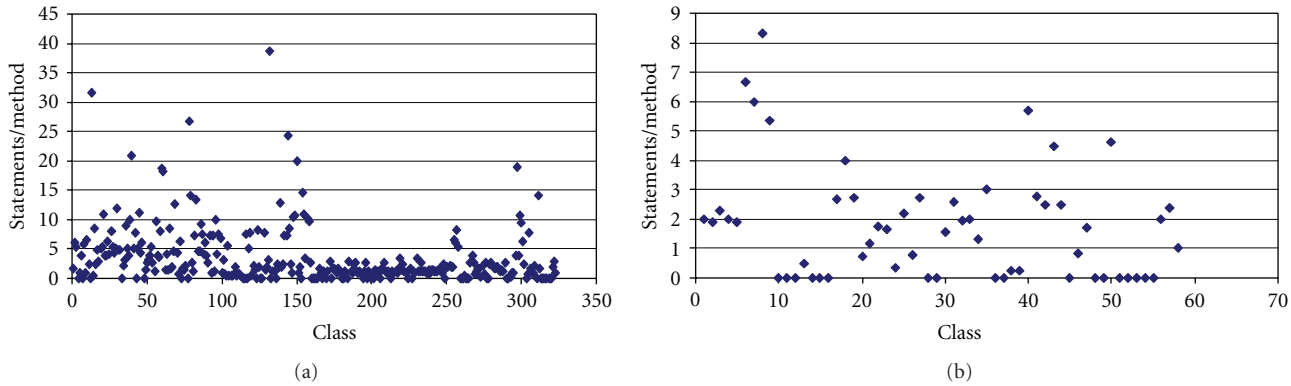


Figure 3: (a) Statements/method (version 1). (b) Statements/method (version $n$).

```
public enum Fulfilment Type
{
None= 9, Site= 11, Ref= 12, BkrRef= 13
}
```

Figure 4: Enumerated Type.

28%) of the 401 classes in total. For Figure 3(b), 80% of values were found in 23 (approximately 22%) of the 101 classes, emphasizing the change in distribution of the values in the later version and the need for fewer statements per method in the system as a whole. This is supported through the standard deviation with values in Figure 3(a) of 4.82, compared with 1.92 for Figure 3(b).

*5.2.3. Lazy Class.* If we now turn to the Lazy Class smell, one route to its identification would be to order classes on statements (ascending) and work downwards thereafter. If we order version 1 in this way, we find that 16 classes had just a single statement. Many of these classes were single, type-based classes similar to that shown in Figure 4. Many other small classes with 3 statements were collection-based classes (38 classes fell into this category) which returned values from a collection, based on a parameter index value passed. In other words, each of these had a specific and cohesively tight functionality and would not, at first impressions, be candidates for amalgamation into other classes.

The complexity of such classes was also found to be slightly lower than average (typically 1.67 compared with 1.70 overall). Again as per the Large Class smell, we note a conflict between the need for a class to remain cohesive (and hence reasonably small) and the search for lazy (small) classes. Put another way, detecting lazy classes may be flawed by definition. For version $n$, the class with smallest number of statements was `ProductDetailsModel.cs`, comprising 2 statements; this is an empty class. The next smallest is class `LoansServiceAction.cs` which contains 3 statements comprising an enumerated data type only. Interestingly, 3 of the 10 smallest classes are "view" interfaces and not classes.

*5.2.4. Comments.* One strategy for identifying classes with large numbers of comments would be to list all classes on ascending percentage of comment lines and examine the code in the resultant classes. By doing this for version 1, we find that four classes all comprise exactly 69% comments. Inspection of the code revealed that none of these classes contained any methods or any other features associated with the OO paradigm. Each of these classes in fact revealed no executable C# code. The files contained information on assembling .net version information and standards for creating new versions of code. The files contained mostly comment lines surrounding that assembly information. The class with the next highest number of comment lines was a class with 55% of comments and represented the initial page requirements for the ITWeb application. For example, the questions that a user had to answer before receiving a quote were an integral part of this class. Intenestingly, inspection of the code in the class revealed old code to have been commented out rather than deleted. In other words, the high percentage of comments was not attributed to comments in the true spirit of the word, but to code that was no longer needed. There were only seven "live" methods in the class, with very few comments associated with these methods.

For version $n$, the maximum percentage of comments amongst the set of 101 classes was just 5.1%, and this belonged to a class called `ServiceErrorHandler.cs`. Inspection of the code for this class showed a very sparse distribution of comments, as might be expected from the percentage of comments it contains and the aggressive nature of the refactoring for the system as a whole. Only 10 of the 101 classes had a percentage number of comments above zero, and seven of those ten classes contained less than 1% comments. It appears that the process of refactoring (from version 1 to $n$) caused a dramatic reduction in the level of comments in the subsystem. All "commented out" lines of code had also disappeared from the system. It is also interesting that 91 classes in version $n$ had zero comments. Discussions with the project leader suggest that the refactored subsystem had become a stable part of the overall system, experienced very few faults, and thus required little maintenance. Figure 5(a) shows the percentage comments for the classes in version one and in Figure 5(b) the same distribution for version $n$ of the same system. The contrast between the range of comments in version 1 and

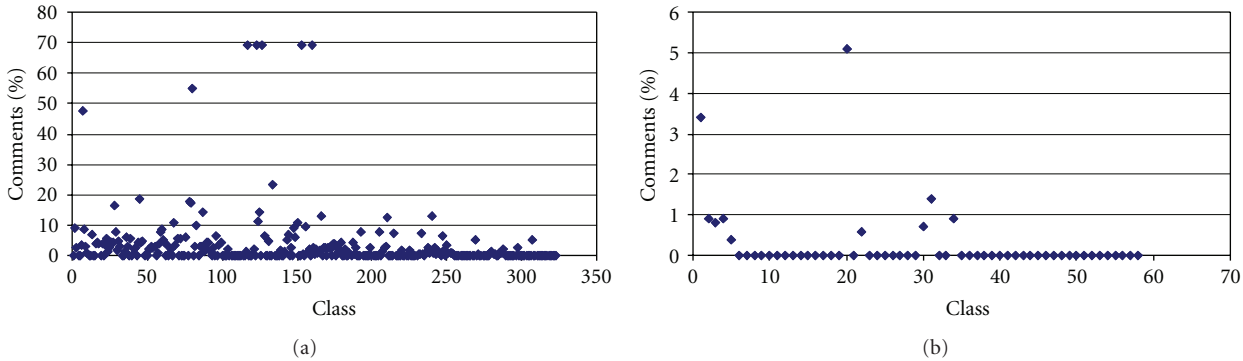


Figure 5: (a) Percentage comments (version 1). (b) Percentage comments (version *n*).

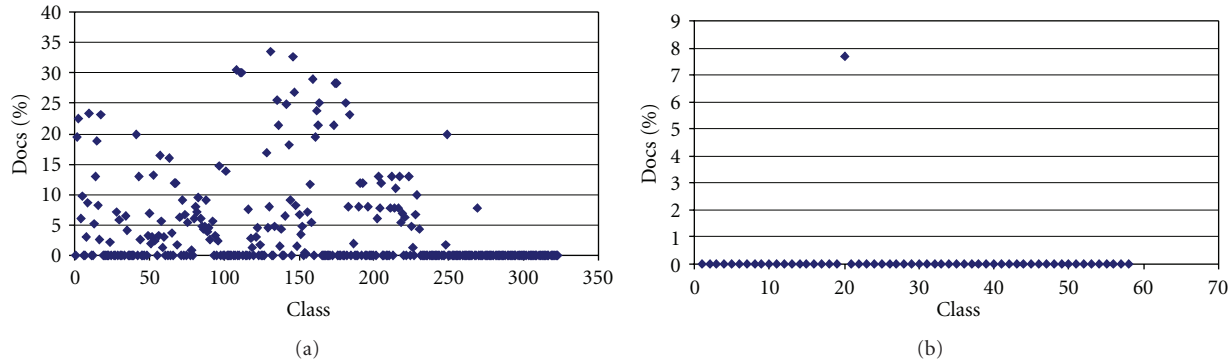


Figure 6: (a) Percentage Docs (version 1). (b) Percentage Docs (version *n*).

version *n* is clear from the two figures in terms of the low percentages in Figure 5(b).

We conducted an 80/20 analysis of the values in each of Figures 5(a) and 5(b) to determine the distribution changes that had taken place from version 1 to *n*. For Figure 5(a), 80% of values were found in 56 (approximately 14%) of the 401 classes in total. For Figure 5(b), 80% of values were found in 6 (approximately 10.34%) of the 101 classes. A marginally shorter head and a longer tail are therefore evident in Figure 5(b). However, that said, the standard deviation of the values in Figure 5(a) is 9.76, compared with just 0.83 for Figure 5(b). This result for comments illustrates that as a result of aggressive refactoring, fewer comments were needed in the refactored version of the system; the standard deviation values suggest that the variation in numbers of comment lines in classes was also less as a result. In other words, one side-effect of refactoring might have been less complex code and the need for fewer comments in fewer classes as a result.

It would appear that, of the four smells that we have considered, reduction in number of comments may be a clear "byproduct of aggressive refactoring effort. It would also seem that addition of comments (based on the high number of zero-values from Figure 5(b)) was not a consideration for the refactored subsystem. We also have to consider the "doc" comments in each version (these are automatically created through the inclusion of XML tags in the source code). The compiler will search for all XML tags in the source code and create an XML documentation file accordingly. Figure 6(a) shows the percentage of "Doc" comment lines in the classes of version 1 and Figure 6(b) that for version *n*. To complete our analysis, we analysed the occurrence of "Doc" comments on an 80/20 basis for Figures 6(a) and 6(b). For version 1, 66 of classes comprised 80% of the percentage Docs (approximately 16%) of the 401 classes. For version *n*, 1 single class contained 100% of all Docs as can be seen from Figure 6(b). All remaining 100 classes comprised zero Docs. The result reflects the nature of the refactoring effort invested and the reduction in both percentage comments (Figures 5(a) and 5(b)) and Docs that the aggressive refactoring policy induced.

## 6. Threats to Validity

Several criticisms or threats could be leveled at the study. First, the analysis presented makes the assumption that a developer has no sense of the effort required to eradicate a smell, in other words, that a developer is oblivious to the presence of smells, and their eradication. This stance might be naïve on our part. A developer might be able to detect a smell; the same developer might also be well aware of the relative advantages of leaving that smell to become a "stench" or to eradicating that smell. Developers also have to make difficult choices as to how they allocate their time. We have no evidence that developers actively avoid smells. Second, there may be many other types and flavors of code smell that

a developer would consider eradicating before those listed in Table 1. We cannot assume that the 22 smells listed in Fowler [1] are necessarily the definitive set of smells. That said, we see this work as a worthwhile start to establishing the parameters through which code smells on a longitudinal basis [32] can be properly and fully explored. Third, we cannot discount the extent to which in the empirical studies presented tools were used to identify the source of code smells. In the case of the ITWeb subsystem, we know of no tools used in the process of smell eradication. Future work will focus on exploring two issues: we intend exploring the test implications of smells and second, on validating the practical and theoretical results with industrial developers and on a range of open-source systems [33]. Finally, as an emerging and developing area, we see potential for a systematic code smell literature review to assess the state-of-the-art and establish key research themes.

## 7. Conclusions/Future Work

In this paper, we have described three studies based on code smells, two of an empirical nature and one a theoretical analysis which attempts to place those studies in context. First, we provided a theoretical suggestion as to why eradicating smells might be problematic. The second study tried to establish whether developers eradicated smells based on the refactorings applied to five open-source systems. The final study was of a proprietary C# system where aggressive refactoring had taken place and a large reduction in system size had been experienced. The key findings of the study presented are as follows; first, no evidence of widespread eradication of smells was found; evidence of simple refactorings in favor of more "complex" refactorings was found. Developers might either be unaware that they are actually eradicating smells or simply avoid complex smells (if empirical data extracted from systems is used as a guide). Second, a wide range of conflicts, contradictions, and anomalies soon emerge when we first try to identify code that smells. This makes the task of identifying "real" smells problematic and possibly prohibitive. Finally, the projected effort in terms of related refactorings that need to be undertaken to eradicate a smell may also be a prohibiting factor. We urge more studies on smells, and to assist in a small way, the data from studies 1 and 3 can be made available upon request of the lead author.

## Acknowledgments

The authors thank the anonymous reviewers for their comments which have contributed significantly to improvements in the paper. The research in this paper was supported by a grant from the UK Engineering and Physical Sciences Research Council (EPSRC) (Grant no.: EP/G031126/1).

## References

[1] M. Fowler, *Refactoring (Improving the Design of Existing Code)*, Addison Wesley, 1999.

[2] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar, "Common refactorings, a dependency graph and some code smells: an empirical study of java OSS," in *Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE '06)*, pp. 288–296, Rio de Janeiro, Brazil, September 2006.

[3] S. Counsell, "Is the need to follow chains a possible deterrent to certain refactorings and an inducement to others?" in *Proceedings of the 2nd International Conference on Research Challenges in Information Science (RCIS '08)*, pp. 111–122, Marrakech, Morocco, June 2008.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, pp. 166–177, Minneapolis, Minn, USA, October 2000.

[5] J. Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2005.

[6] T. Mens and A. van Deursen, "Refactoring: emerging trends and open problems," in *Proceedings 1st International Workshop on REFactoring: Achievements, Challenges, and Effects (REFACE '03)*, Univ. of Waterloo, 2003.

[7] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[8] R. Najjar, S. Counsell, G. Loizou, and K. Mannock, "The role of constructors in the context of refactoring object-oriented software," in *IEEE European Conference on Software Maintenance and Reengineering*, pp. 111–120, Benevento, Italy, March 2003.

[9] W. Opdyke, *Refactoring object-oriented frameworks*, Ph.D. thesis, Univ. of Illinois, 1992.

[10] M. V. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad smells—humans as code critics," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pp. 399–408, Chicago, Ill, USA, September 2004.

[11] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: an empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, 2006.

[12] M. V. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions," in *Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISCE '06)*, pp. 297–306, Rio de Janeiro, Brazil, September 2006.

[13] M. Mäntylä, *Bad smells in software–a taxonomy and an empirical study*, M.S. thesis, Helsinki University of Technology, Software Business and Engineering Institute, 2003.

[14] F. Khomh, S. Vaucher, Y.-G. Guéehéeneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings 9th International Conference on Quality Software (QSIC '09)*, C. Byoung-ju, Ed., pp. 305–314, August 2009.

[15] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings 16th Working Conference on Reverse Engineering (WCRE '09)*, pp. 75–84, Antwerp, Belgium, October 2009.

[16] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: a case study of two open source systems," in *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, pp. 390–400, Lake Buena Vista, Fla, USA, October 2009.

[17] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *Proceedings of IEEE International Conference on Software Maintenance (ICSM '10)*, Timisoara, Romania, September 2010.

[18] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[19] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE '02)*, Richmond, Va, USA, 2002.

[20] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pp. 350–359, Chicago, Ill, USA, September 2004.

[21] R. Marinescu, "Measurement and quality in object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 701–704, September 2005.

[22] H. Hamza, S. Counsell, G. Loizou, and T. Hall, "Code smell eradication and associated refactoring," in *Proceedings of the European Computing Conference (ECC '08)*, Malta, September 2008.

[23] S. Counsell and S. Swift, "Refactoring steps, java refactorings and empirical evidence," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, pp. 176–179, Turku, Finland, August 2008.

[24] SourceMonitor Tool, http://www.campwoodsw.com/sourcemonitor.html.

[25] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, International Thomson Press, London, UK, 2002.

[26] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[27] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 2004.

[28] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2003.

[29] S. Counsell, S. Swift, and J. Crampton, "The interpretation and utility of three cohesion metrics for object-oriented design," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 2, pp. 123–149, 2006.

[30] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 45–54, Amsterdam, The Netherlands, 2003.

[31] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," in *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP '06)*, vol. 4044 of *Lecture Notes in Computer Science*, pp. 75–84, Oulu, Finland, June 2006.

[32] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 493–509, 1999.

[33] T. T. Dinh-Trong and J. M. Bieman, "The FreeBSB project: a replication case study of open source development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 481–494, 2005.

Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 273080, 9 pages
doi:10.1155/2010/273080

*Research Article*

# On the Use of Issue Tracking Annotations for Improving Developer Activity Metrics

**Andrew Meneely and Laurie Williams**

*North Carolina State University, 890 Oval Drive, Engineering Building 2, Room 3272, Campus Box 8206, Raleigh, NC 27695-8206, USA*

Correspondence should be addressed to Andrew Meneely, apmeneel@ncsu.edu

Received 5 September 2010; Revised 2 December 2010; Accepted 31 December 2010

Academic Editor: Giulio Concas



Understanding and measuring how teams of developers collaborate on software projects can provide valuable insight into the software development process. Currently, researchers and practitioners measure developer collaboration with social networks constructed from version control logs. Version control change logs, however, do not tell the whole story. The collaborative problem-solving process is also documented in the issue tracking systems that record solutions to failures, feature requests, or other development tasks. We propose two annotations to be used in issue tracking systems: solution originator and solution approver. We annotated which developers were originators or approvers of the solution to 602 issues from the OpenMRS healthcare system. We used these annotations to augment the version control logs and found 47 more contributors to the OpenMRS project than the original 40 found in the version control logs. Using social network analysis, we found that approvers are likely to score high in centrality and hierarchical clustering. Our results indicate that our two issue tracking annotations identify project collaborators that version control logs miss. Thus, issue tracking annotations are an improvement in developer activity metrics that strengthen the connection between what we can measure in the project development artifacts and the team's collaborative problem-solving process.

## 1. Introduction

The quality of many software products rests on teams of people who are collaborating with each other. Measuring how teams of developers collaborate on software projects can provide valuable insight into the software development process. One class of metrics, called *developer activity metrics*, analyzes the structure of a development team by quantifying how developers collaborate with each other [1]. Already, developer activity metrics have been shown to predict failures [2, 3], predict vulnerabilities [1], and provide insight on individual projects [4–7].

Many studies using developer activity metrics use version control change logs to determine who is working on which part of the system. Version control change logs, however, do not tell the whole story. While a version control system records the final solution to an issue, the elements of how that solution came to be are captured in other artifacts, such as the issue tracking system. In issue tracking systems, developers can take a known issue (e.g., a failure or a feature request) and assign it a "ticket". Developers can then link pertinent artifacts to that ticket, such as patches, change sets in the version control system, error logs, or screenshots. Also on the ticket is an online discussion of how to resolve the issue.

Consider the following scenario (taken from http://dev.openmrs.org/ticket/1171). A large open source project has an open ticket that needs resolving. A user named Frank then submits a patch to fix the problem by linking his patch to the ticket. An online discussion amongst the system's developers and users then takes place, which ends in developer Ben deciding that Frank's solution is correct. Ben then applies the changes from Frank's patch, linking the version control change set to the ticket. The version control change logs would show that only Ben has fixed the problem when, in fact, Frank *originated* the solution, while Ben *approved* the

solution. That the two developers collaborated on a solution ought to be captured in developer activity metrics.

We propose two issue tracking ticket annotations: solution originators and solution approvers. Intended to "give credit where credit is due", these annotations can be used to augment the logs from version control systems to provide more accurate information about who contributed to which parts of the system.

The objective of this research is *to improve the information gained by measurements of developer collaboration by introducing and analyzing two issue tracking annotations: solution originator and solution approver*. Our aim is to evaluate the usefulness of the annotation in terms of discovering collaborators in a software development project. We examined the online discussions of 602 tickets from the OpenMRS (http://openmrs.org/) healthcare web application issue tracking system, annotating which developers were originators and/or approvers of the solution to the ticket.

We performed an empirical analysis of how much information was gained by using issue tracking annotations in combination with version control change logs. Forming social networks of developers based on our data, we examined correlations between the annotations and several social network analysis metrics. Our social network analysis metrics come from centrality and hierarchical clustering techniques.

The rest of the paper is organized as follows. Section 2 describes background and related work with respect to developer activity metrics, centrality, developer networks, and issue tracking systems. Section 3 describes the developer network. Section 4 describes our data collection process. Section 5 describes our analysis and results. Sections 6 and 7 summarize our limitations and conclusions.

## 2. Background and Related Work

As a project progresses, developers make changes to various parts of the system. With many changes and many developers, changes to files tend to overlap: multiple developers may end up working on the same files around the same time, indicating that they share a common contribution, or a *connection*, with another developer. Some developers end up connected to many other highly connected developers, some end up in groups (clusters) of developers, and some tend to stay peripheral to the entire network.

In this paper, we use network analysis to quantify how developers collaborate on projects. Network analysis is the study of characterizing and quantifying network structures, represented by graphs [8]. In network analysis, vertices of a graph are called nodes, and edges are called connections. A sequence of nonrepeating, adjacent nodes is a path, and a shortest path between two nodes is called a geodesic path (note that geodesic paths are not necessarily unique). Informally, a geodesic path is the "social distance" from one node to another.

"*Centrality*" metrics are used to quantify the location of a node relative to the rest of the network. In this study, we use two measures of node centrality: degree and betweenness. The "*degree*" of a node is equal to the number of neighbors a developer has in the network. While the "*degree metric*" is based on direct connections to other developers, the "*betweenness*" metric is based on a developer's indirect connections to the rest of the network. If a developer has many connections, the betweenness [8] of node $n$ is defined as the number of geodesic paths that include $n$. A high betweenness means a high centrality.

"*Clustering*" algorithms are techniques for detecting community structures in social networks. A cluster of nodes is a *set* of nodes such that the number of intraset connections greatly outnumbers the number of interset connections [8]. A cluster of developers, then, has more connections within the cluster than to other developers. Furthermore, since communities can have many layers of subcommunities, "*hierarchical clustering*" techniques detect the clusters within clusters. In this study we use the Girvan-Newman [8] hierarchical clustering algorithm, which applies centrality concepts to edges in the network and iteratively generates a hierarchy of developers according to the number and size of the clusters they belong to.

Also, we use the term "*issue*" to include all potential types of change requests in a system, including failures, vulnerabilities, feature requests, and tasks. When we refer to a "*ticket*," we are referring to the record of a specific issue as kept by the issue tracking system. Tickets can have discussions on them, in a message board fashion. The solution to a ticket is embodied in a "*change set*," which is a set of changes made to a code base kept track of by the version control system.

Much work has been done in the area of measuring developer collaboration on software projects. The applications of these measurements are particularly diverse, ranging from failure and vulnerability prediction to studying open source software projects.

Gonzales-Barahona et al. [6] were the first to propose the idea of creating developer networks as models of collaboration from version control systems. The authors' objective was to present the developer network and to differentiate and characterize projects. Their work did not include any integration with issue tracking systems.

Bird et al. [5] used developer networks to examine social structures in open source projects. Discussing the bazaar-like development of open source projects, the authors empirically examine how open source developers self-organize. The authors use similar network structures as our developer network to find the presence of subcommunities within open source projects. In addition to examining version control change logs, the authors mined email logs to find a community structure. The authors conclude that subcommunities do exist in open source projects, as evidenced by the project artifacts exhibiting a social network structure that resembles collaboration networks in other disciplines.

Pinzger et al. [3] proposed a similar structure to the developer network, called the contribution network. The contribution network is designed to use version control data to quantify the direct and indirect contribution of developers on specific resources of the project. The researchers used metrics of centrality in their study of Microsoft Windows

Vista and found that closeness was the most significant metric for predicting reliability failures. Files that were contributed to by many developers, especially by developers who were making many different contributions themselves, were found to be more failure-prone than files developed in relative isolation. The finding is that files which are being focused on by many developers are more likely to have a failure than files developed by few developers.

Meneely et al. [2] examined the relationship between developer activity metrics and reliability. The empirical case study examined three releases of a large, proprietary networking product. The authors used developer centrality metrics from the developer network to examine whether files are more likely to have failures if they were changed by developers who are peripheral to the network. The authors formed a model that included metrics of developer centrality, code churn (the degree to which a file was changed recently), and lines of code to predict failures from one release to the next. Their model's prioritization found 58% of the system's failures in 20% of the files, where a perfect prioritization would have found 61%. The study did not include integration with issue tracking systems.

Sarma et al. [9] built a tool, called Tesseract, to visualize developer connections within a software project. Designed to build a useful summary of what they call the "network of artifacts" in each software project, Tesseract gathers its information from version control change logs, source code dependency graphs, issue tracking tickets, and developer communication logs. In this particular study, they established that Tesseract was both a usable and useful tool according to the user studies and interviews they conducted.

Nagappan et al. [10] created a logistic regression model for failures in the Windows Vista operating system. The model was based on what they called "Overall Organizational Ownership" (OOW). The metrics for OOW included concepts like organizational cohesiveness and diverse contributions. The authors found that more edits made by many noncohesive developers lead to more problems post release. The OOW model was able to predict with 87% average precision and 84% average recall. The OOW model bears a resemblance to the contribution network by Pinzger et al. [3] in that both models attempt to differentiate healthy changes in software from the problematic changes.

Meneely and Williams [1] applied developer activity metrics to security data in examining aspects of the saying "Many eyes make all bugs shallow" (known as Linus' Law [11]). Using both developer networks and the contribution networks proposed by Pinzger et al. [3], the authors examined several metrics related to developer collaboration, including a clustering metric applied to developer networks. The authors found that files changed by nine or more developers were 16 times more likely to have at least one vulnerability than files changed by fewer than nine developers. Their analysis did not include data from issue tracking systems, only from version control change logs.

Lastly, our original proposal for extending developer networks using issue tracking annotations included an analysis of the OpenMRS system [7].

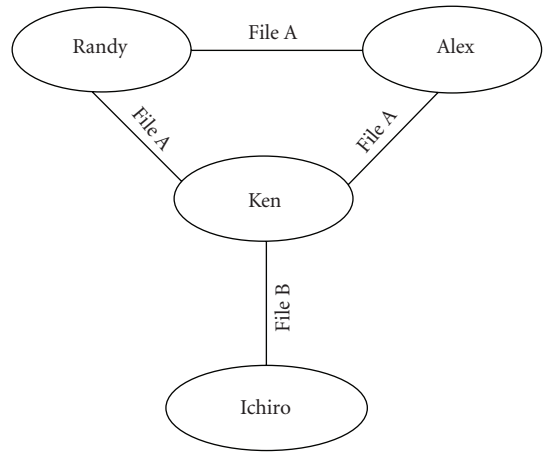


Figure 1: Resulting developer network from Table 1.

Table 1: Example developer contributions.

| Developer | Contributions |
|---|---|
| Alex | File A |
| Ken | File A, File B |
| Randy | File A |
| Ichiro | File B |

The developer network from Table 1 is shown in Figure 1.

## 3. Extending Developer Networks

The developer network is designed to represent the complex structure of development in terms of people. The idea is to infer "who is working with whom" by examining "who is working on the same code". In our developer network, developers are represented as nodes, and edges exist between two nodes where two developers made contributions to the same source code file within one month. As a result, the developer network is an undirected, unweighted, and simple graph.

In prior research [1–3, 5, 6, 10], the existence of a "contribution" to a given source code file was gathered from only version control change logs. We will refer to such a structure as a "*developer network*" (or "regular" developer network).

For example, suppose that we have the contribution table found in Table 1.

In Figure 1, developer Ken has a degree of three. Ken also has a betweenness of five since he is on five of the geodesic paths (three originating from himself). Ichiro has a betweenness of three, while Randy and Alex have a betweenness of four. More examples of developer networks and their usage can be found in other works [1, 2, 5, 9].

In this study, we extend the notion of contributions beyond version control change logs to originators and approvers of solutions in issue tracking systems. Thus, if a person was found to be the originator or approver of the solution to a ticket, that person would be marked as making a contribution to the source code files in the final commit (more information can be found in Section 4).

Therefore, we will refer to the developer network gathered from version control logs *and* our issue tracking annotations as the *extended developer network*.

For this study, we annotated tickets manually by examining issue tickets post hoc, as described in Section 4. However, this data could be gathered earlier and with less labor if the issue tracking system supports a "solution approver" and a "solution originator" field on the ticket. If issue tracking systems allowed for our two annotations, the developers on each ticket could decide who are the solution originators and solution approvers and could form an extended developer network automatically. To our knowledge, no such field exists in issue tracking systems such as Bugzilla (http://www.bugzilla.org/) or Trac (http://trac.edgewall.org/).

## 4. Collecting Annotations

The developers of the OpenMRS project use Trac for their issue tracking system and Subversion (http://subversion.tigris.org/) (SVN) for their version control system. By default, Trac will link a coded change set in SVN to a ticket if the developer uses the hash mark (#) and the ticket number in the Subversion commit message. While this feature is optional, we found that the OpenMRS developers were meticulous about linking change sets to tickets when the issue was resolved. OpenMRS had over 1900 tickets logged during the time that we studied, of which 602 resulted in a change set to resolve the issue.

We manually examined the discussions of those 602 tickets to annotate who on the ticket were the approver and originator of the solution to the issue. We used the following steps for each ticket.

(1) Read the ticket's description to understand the problem.

(2) Read the online discussion directly on the ticket.

(3) Read any discussions in separate forums (e.g., mailing list) linked from the ticket.

(4) Read the SVN comments left by the person who committed the solution to version control.

(5) Compare the patches attached to the ticket to the solution committed to SVN and determine which patches were used in the issue's solution.

(6) Based on the information gained in steps (1)–(4), decide who the originators are and who the approvers are.

The first author and an additional researcher executed the latter six steps independently. Both researchers then compared annotations and resolved each disagreement in annotation.

We considered a person to be an originator if

(i) a person submitted a patch that was a major part of the solution to the issue, or

(ii) a person introduced code (e.g., wrote code directly into the ticket's message board), and that code was used in the solution, or

(iii) the committer of the solution attributed someone on the ticket in their Subversion commit message.

We considered a person to be an approver if

(i) the person changed the ticket status to "approved", or

(ii) the person, in the course of the discussion, made the final decision as to what ought to be done in the code (but did not necessarily enact the change in the code), or

(iii) the person applied the patch and closed the ticket.

One ticket could have multiple originators and multiple approvers. Not included in our annotations are people who made contributions to the ticket discussion but did not participate in the final solution. Furthermore, a person could be both an originator and an approver to a solution if they created a public ticket and then resolved it themselves. Lastly, a person could be neither an originator nor an approver but still make contributions to the code if they only committed code directly to version control system without participating in any solutions to public tickets. We call these people "contributors".

If a person was an originator or approver, they were recorded as making a contribution to the code affected by the solution at the time the solution was applied in addition to the original committer to the version control system. With the records of contributions from contributors, approvers, and originators, the developer network was calculated as described in Section 2. As a result of this construction, originators and approvers of the same ticket are always connected to each other. This automatic connection matches the notion of a collaboration connection since originators and approvers are collaborating on a solution.

## 5. Evaluation

In this section, we examine whether information about developer collaboration can be gained by analyzing two issue tracking annotations: solution originator and solution approver. First, in Section 5.1, we examine the developers found by applying issue tracking annotations. Next, in Sections 5.2 through 5.5, we evaluate the following scientific hypotheses.

$H_1$: *Approvers have a higher developer network centrality than nonapprovers.*

$H_2$: *Originators have a higher developer network centrality than nonoriginators.*

$H_3$: *Central developers in regular developer networks are also central in extended developer networks.*

$H_4$: *Approvers belong to more clusters than nonapprovers.*

*5.1. Analyzing Information Gained from Annotations.* First, we must ask if our anecdotal observations of SVN logs are true: do the originator and approver annotations provide more information that was not gained from analyzing the version control change logs? Perhaps the commits from the

Table 2: Approvers in regular and extended developer networks.

| | Only in regular DN | Only in extended DN | Total |
|---|---|---|---|
| Nonapprover | 24 (60%) | **39 (82%)** | 63 (72%) |
| Approver | 16 (40%) | 8 (18%) | 24 (27%) |
| Total | 40 (100%) | 47 (100%) | 87 (100%) |

Table 3: Originators in regular and extended developer networks.

| | Only regular DN | Only in extended DN | Total |
|---|---|---|---|
| Nonoriginator | 13 (32%) | 7 (15%) | 20 (23%) |
| Originator | 27 (68%) | **40 (85%)** | 67 (77%) |
| Total | 40 (100%) | 47 (100%) | 87 (100%) |

version control system are enough to identify the entire structure of the team.

A visual comparison of the developer network and extended developer network can be found in Figure 2 (shown in the appendix). The developer network more than doubled in size when including annotations. The developer network turned out to have 40 developers, while the extended developer network had 87 developers. As a result, our annotations identified 47 developers who did not make SVN commits but contributed to a solution adopted into the project.

Furthermore, the majority of the 47 developers found only in the extended network were originators and not approvers. Tables 2 and 3 show how the developer counts break down in terms of annotations in each network. Had one used a developer network from only version control logs, one might conclude that the development team is 40% approvers, when in fact there are fewer approvers (27%).

The high rate of nonapprovers is found only in the extended network (cell highlighted in gray). Also, Table 3 shows a high rate of originators found only in the extended network (cell highlighted in gray).

Additionally, the complexity of the network structure increased dramatically. Figure 2 (shown in the appendix) visually demonstrates the difference between the two networks rather starkly. Both networks have the same layout; so developers in both networks are in the same location in each diagram.

Therefore, when comparing the regular developer network to the extended developer network, the number of contributors to solutions adopted by the OpenMRS project more than doubles in size because many originators and nonapprovers are being accounted for.

*5.2. Approver Centrality.* Developer network centrality is a measure of how directly and indirectly a developer is to the rest of the network. When a developer has a high centrality, then he or she has worked on files with many other people. In previous work [2], we found that developer centrality can be used for predicting failures in files.

Table 4: Approver centralities.

| Metric | Nonapprover mean | Approver mean | MWW $P$-value $< .01$? | Power |
|---|---|---|---|---|
| Degree | 6.0 | 16.9 | Yes | 0.81 |
| Betweenness | 2.7 | 132.6 | Yes | 0.99 |

Furthermore, we observed that developers we annotated as solution approvers tended to be people who were well-known and knowledgeable enough to be trusted with approving solutions to problems. Therefore, we hypothesize that developer centrality is correlated with being an approver.

$H_1$: *Approvers have a higher developer network centrality than nonapprovers.*

We evaluate $H_1$ by evaluating the following null and alternative hypotheses.

$H_{1(0)}$: *Approvers have the same developer network centrality as nonapprovers.*

In this paper, we use two measures of centrality: degree and betweenness (defined in Section 2). We used the Mann-Whitney-Wilcoxon (MWW) test (at $P < .05$) to examine the differences in centrality between approvers and nonapprovers. We chose the nonparametric MWW because it does not assume that developer centralities are normally distributed. To evaluate which group has the higher centrality, we compared the means. Since we are evaluating multiple hypotheses, we used a Bonferroni correction; so we check our $P$-values against .01 instead of the traditional .05. Lastly, we perform a power analysis to ensure that our sample groups are large enough and compare the results against the traditional 8. Table 4 shows our results.

For both metrics, the approvers had a higher developer centrality than nonapprovers; thus we reject the null hypothesis $H_{1(0)}$. Therefore, central developers of the extended developer network are also approving solutions to the issue tickets. Figure 3 (shown in the appendix) visually illustrates that the approvers are generally well-connected developers in the network.

*5.3. Originator Centrality.* Centrality is related not only to *how many* connections a developer has, but also *to whom* a developer is connected (e.g., if a developer is connected to a very central developer, she becomes more central herself). Since originators and approvers on the same ticket are automatically connected to each other, then if approvers are central, then originators are also central. Thus, we examine the following hypothesis.

$H_2$: *Originators have a higher developer network centrality than nonoriginators.*

The null version of this hypothesis is as follows.

$H_{2(0)}$: *Originators have the same developer network centrality as nonoriginators.*

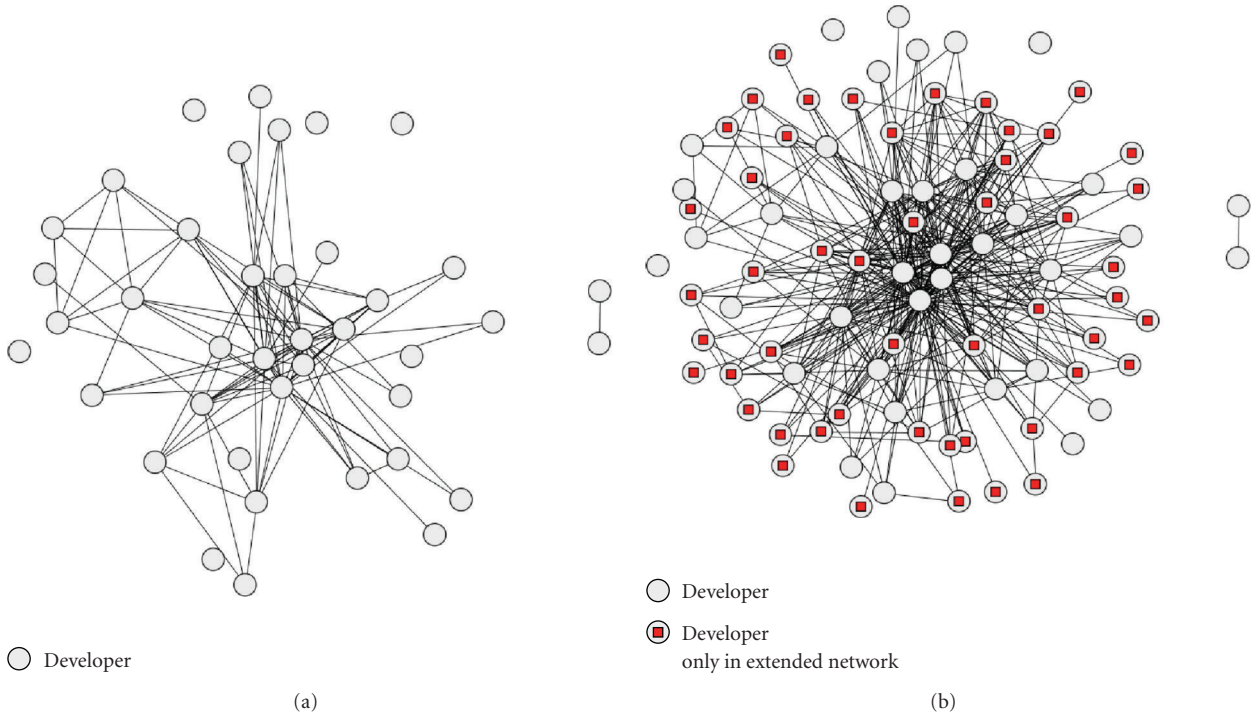


Figure 2: (a) Regular developer network and (b) extended developer network of the same layout.

We applied the same analysis as in the previous section. Our results are in Table 5.

Again, for both metrics, the originators had a higher centrality than nonoriginators; thus we reject the null hypothesis $H_{2(0)}$.

One may also notice that the difference in both betweenness and degree is not as drastic as with approvers in Table 4. This result also aligns with our motivation for this hypothesis: originators are central because they are connected to approvers, who are also central.

*5.4. Developer Networks and Extended Developer Networks.* Although we concluded in Section 5.1 that annotating the issue tracking system provides valuable information, this situation is not always feasible. Issue tracking annotations require either participation by the development team or manual inspection post hoc (as in this study). But, the regular developer network may *resemble* the extended network enough to still be useful.

One resemblance between the two networks could be the relative developer centralities. That is, are the central developers of the developer network central to the extended developer network? Thus, we test the following hypothesis.

$H_3$: *Central developers in regular developer networks are also central in extended developer networks.*

The null version of this hypothesis is as follows.

$H_{3(0)}$: *Central developers in regular developer networks are not necessarily central in extended developer networks.*

Table 5: Originator centralities.

| Metric | Nonoriginator mean | Originator mean | MWW $P$-value $< .01$? | Power |
|---|---|---|---|---|
| Degree | 2.7 | 10.9 | Yes | 1.0 |
| Betweenness | 1.2 | 49.7 | Yes | 1.0 |

For this analysis, we use the nonparametric Spearman rank correlation coefficient between the developer centralities. We use a statistical test based on rank because the scales of developer centrality differ. For example, a degree of 10 may be considered high in one network and low in another. Centralities of developers that were only in the extended developer network were excluded from this test. We report the correlation coefficient, along with its statistical significance, in Table 6.

These correlations are fairly strong, indicating that developer centrality of an extended network is good estimators of the developer centrality of an extended network. Thus, we reject hypothesis $H_{3(0)}$.

With these correlations, we investigated further into whether or not the developer centrality from a regular developer network is correlated with being an approver or not. We used the Mann-Whitney-Wilcoxon test again to see if there are any differences in the centrality of the regular developer network between approvers and nonapprovers. Our results for approvers can be found in Table 7.

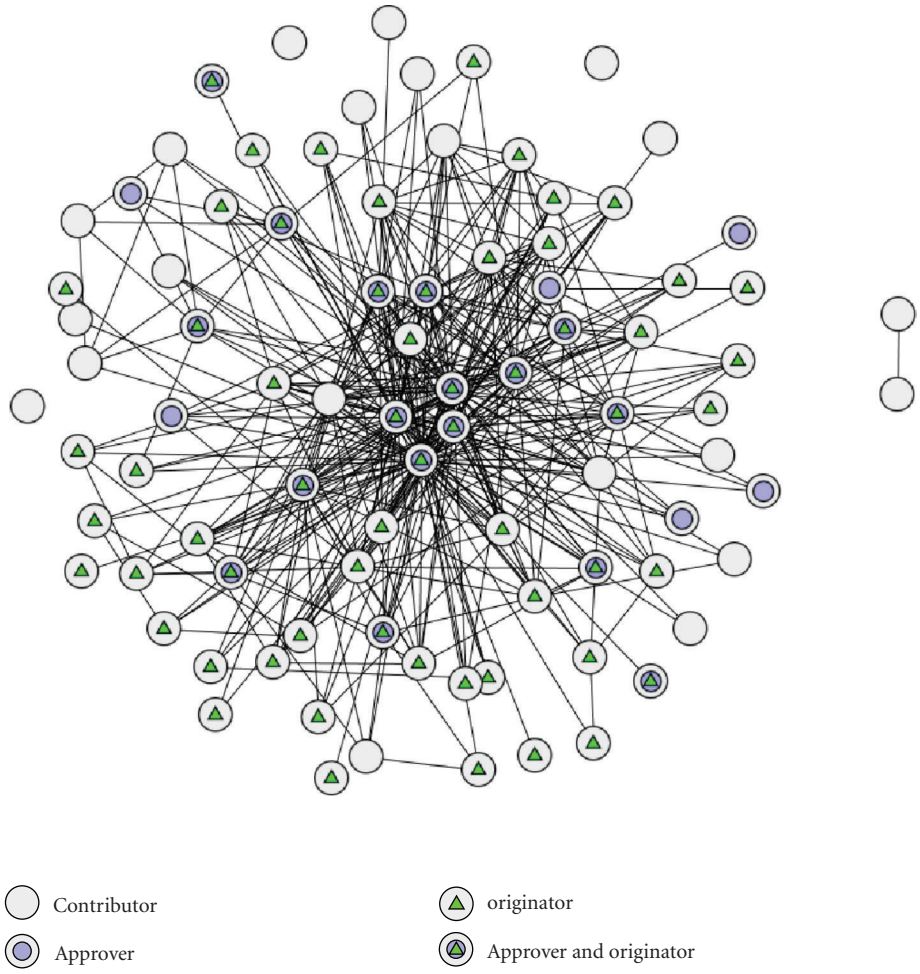


Figure 3: Extended developer network with annotations.

Table 6: Correlation between centralities of regular and extended developer network.

| Metric | Spearman between regular and rxtended | $P$-value < .01? |
|---|---|---|
| Degree | 0.67 | Yes |
| Betweenness | 0.85 | Yes |

Table 7: Approver centralities, regular developer network.

| Metric | Nonapprover mean | Approver mean | MWW $P$-value < .01? | Power |
|---|---|---|---|---|
| Degree | 1.9 | 9.5 | Yes | 1.0 |
| Betweenness | 0.3 | 29.2 | Yes | 1.0 |

For both metrics, developer centrality from the regular developer network was higher for approvers than for nonapprovers.

We also investigated whether or not central developers in the regular developer network are also originators. Our results can be found in Table 8.

Table 8: Originator centralities from regular developer network.

| Metric | Nonoriginator mean | Originator mean | MWW $P$-value < .01? | Power |
|---|---|---|---|---|
| Degree | 2.15 | 6.2 | No | 1.0 |
| Betweenness | 0.5 | 17.3 | Yes | 1.0 |

While not enough empirical evidence exists to say that the Degree measurements were different for originators, there is enough statistical evidence to say that the Betweenness was different for originators.

As with our results in Sections 5.2 and 5.3, the differences in centralities were not as great for the originators as the differences for the approvers. This result provides more evidence that approvers are the most central developers, and originators are central because they are connected to approvers.

*5.5. Are Approvers in More Clusters?* Expert experience [11] and empirical studies [4–6] have shown that open source communities (e.g., Linux (http://www.linux.org/), Apache

Table 9: Cluster ranks of approvers and nonapprovers from extended developer networks.

| | Approvers | Nonapprovers | MWW $P$-value $< .01$? | Power |
|---|---|---|---|---|
| Mean cluster rank | 28.3 | 42.1 | Yes | 0.86 |

(http://httpd.apache.org/), and PosgreSQL (http://www.postgresql.org/)) tend to self-organize into smaller subcommunities in large development efforts. However, with many small subcommunities, developers must also coordinate all of those development efforts.

One of the techniques for detecting communities in social networks is the Girvan-Newman hierarchical clustering algorithm (described in Section 2). The output of the Girvan-Newman algorithm is a hierarchy of developers according to the number and size of the clusters they belong to.

We hypothesize that if a developer has enough authority in the project to be a solution approver, then that developer also belongs to more clusters. Thus, we evaluate the following hypothesis.

$H_4$: *Approvers belong to more developer subcommunities than nonapprovers.*

The null version of this hypothesis is as follows.

$H_{4(0)}$: *Approvers belong to as many developer subcommunities as nonapprovers.*

We perform our evaluation by running the Girvan-Newman hierarchical clustering algorithm on both our annotated developer network and regular developer network. For each developer in the network, we assign an integer rank (which we call *cluster rank*) according to the output of the clustering algorithm. A rank of one indicates that the developer is in more clusters than any other developer.

To evaluate whether being approver is statistically associated with having a high cluster rank, we used the Mann-Whitney-Wilcoxon test between approvers and nonapprovers in the extended developer network. We also report the mean rank of each population. Our results can be found in Table 9.

The difference in cluster ranks between approvers and nonapprovers was statistically significant. Therefore, we reject the null hypothesis $H_{4(0)}$ that approvers are as likely to be in multiple developer subcommunities than nonapprovers.

Interestingly, the lowest rank we observed for an approver was 56. In that situation, the approver only approved one ticket that was closely related to her own work. Other low-ranking approvers were in similar situations where the authority for approving a ticket was (implicitly or explicitly) given to them based on the scope of the work itself. These results indicate that the authority to approve a few tickets may not be related to the overall authority of the project.

## 6. Discussion

In this section, we discuss some of the ramifications of our results, including relation to prior work and to future work.

*6.1. Are Regular Developer Networks Invalidated?* Our results from Tables 2 and 3 in Section 5.1 show that the version control change logs do not reflect the entire OpenMRS developer community, and that the issue tracking system does contain information about more developers. These results, however, do not contradict nor invalidate previous studies relying only on version control change logs. Our findings in Section 5.4 show that regular developer networks closely resemble the extended developer network in terms of developer centrality.

Instead, the results of this paper shed more light on who central developers are in open source projects: they tend to be people who are contributing solutions to code that other people are also working on. In the past, experts [11] have observed this concept from experience. While open source communities might be perceived as crowds of people all coding at once, most open source communities have a core group of developers who must guide the direction of the project one issue ticket at a time.

*6.2. Why Not Include All Comments?* We do not recommend applying all of the comment entries from the issue tracking system to the developer network. In examining the issue tracking system, we found many unrelated comments in the discussion. For example, sometimes people would comment on a ticket with similar (but not identical) issue. That issue would be forked into a new ticket and is unrelated to the original ticket. Or, some people would not provide helpful, constructive comments that would lead to a tangible contribution to the issue. In our analysis, we specifically chose the originator and approver annotations as noise-reducing mechanisms. Using many e-mail archives or other collaboration artifacts also has this kind of problem.

The process of annotating a historical record, however, is manual and can be labor-intensive. One approach is to apply text mining or other machine learning techniques to automatically identify approvers and originators on a ticket; however such a technique would also be noisy. We believe that the open source community should employ a convention of attributing the originator and approver of a given solution to a ticket.

The Linux kernel (http://git.kernel.org/) is one such community that curates its own data with respect to originators and approvers. Their version control system separates the concepts of "committer" and "author" for a given source code change, which was the inspiration for our "originator" annotation. Furthermore, the Linux kernel community has the convention of having a "Signed-off by" field in their version control commit comments that closely resembles our notion of "approver".

## 7. Limitations

The scope of this study is only on the technical practice of providing coded solutions (via version control or patches). Many other non-developers exist in open source projects that are not being captured by the development artifacts we are studying.

Additionally, our analysis only includes one case study of an open source project. We chose the OpenMRS project as a production-level system with an active community; so we believe that OpenMRS is a good representative of the open source development community. Our techniques would need to be applied to other software development projects to achieve more generality. Also, the manual process of annotating issues could introduce errors in our data set. To mitigate this factor, we had both researchers perform the annotation separately and then resolve any differences. Lastly, the results of this study are correlations; so we cannot claim any causal direction. For example, we do not know if being a central developer causes being an approver, or vice versa.

## 8. Conclusion

The objective of this research is to improve the information gained by measurements of developer collaboration by introducing and analyzing two issue tracking annotations: solution originator and solution approver. We found that applying issue tracking annotations revealed more developers than found in the version control logs. The additional results of our study are as follows.

(i) Approvers have a higher developer network centrality than nonapprovers.

(ii) Originators have a higher developer network centrality than nonoriginators.

(iii) Developer networks without issue tracking annotations resemble developer networks with issue tracking annotations in terms of developer centrality.

(iv) Approvers belong to more developer subcommunities than nonapprovers.

While using annotations captures more information about developer collaboration, we found that regular developer networks are still accurate representations of the development community. This result indicates that a developer network can be used as an estimate for developer collaboration and is, therefore, useful in situations where annotations are not available. Our results are an improvement in developer activity metrics that strengthens the connection between what we can measure in the project development artifacts and the team's collaborative problem-solving process.

## Appendix

See Figures 2 and 3.

## Acknowledgments

The authors thank Mackenzie Corcoran for her contributions to the data collection. This research is supported by the Army Research Office managed by the North Carolina State University Secure Open System Initiative (SOSI). They also thank the OpenMRS development community for opening their data sets to be analyzed.

## References

[1] A. Meneely and L. Williams, "Secure open source collaboration: an empirical study of Linus' law," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pp. 453–462, Chicago, Ill, USA, November 2009.

[2] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM International Symposium on the Foundations of Software Engineering (SIGSOFT '08)*, pp. 13–23, Atlanta, Ga, USA, November 2008.

[3] M. Pinzger, N. Nagappan, and B. Murphy, "Can developer-module networks predict failures?" in *Proceedings of the 16th ACM International Symposium on the Foundations of Software Engineering (SIGSOFT '08)*, pp. 2–12, Atlanta, Ga, USA, November 2008.

[4] C. Bird, A. Gourley, P. Devanbu et al., "Mining email social networks in postgres," in *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR '06)*, pp. 185–186, Shanghai, China, 2006.

[5] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proceedings of the 16th ACM International Symposium on the Foundations of Software Engineering (SIGSOFT '08)*, pp. 24–35, Atlanta, Ga, USA, November 2008.

[6] J. M. Gonzales-Barahona, L. Lopez-Fernandez, and G. Robles, "Applying social network analysis to the information in CVS repositories," in *Proceedings of the International Workshop on Mining Software Repositories (MSR '05)*, pp. 1–8, Edinburgh, UK, 2005.

[7] A. Meneely, M. Corcoran, and L. Williams, "Improving developer activity metrics with issue tracking annotations," in *Proceedings of the Workshop on Emerging Trends in Software Metrics (WETSoM '10)*, pp. 75–80, Cape Town, South Africa, 2010.

[8] U. Brandes and T. Erlebach, *Network Analysis: Methodological Foundations*, Springer, Berlin, Germany, 2005.

[9] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: interactive visual exploration of socio-technical relationships in software development," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pp. 23–33, May 2009.

[10] N. Nagappan, B. Murphy, and V. R. Basili, "The influence of organizational structure on software quality: an empirical case study," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 521–530, Leipzig, Germany, May 2008.

[11] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly and Associates, Sebastopol, Calif, USA, 1999.