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:

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

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) γ_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 κ 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

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

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

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

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) α_k, the average fraction of time that each contributor (administrator or developer) devotes to the project;

(iii) β_k, the percentage of active administrators and developers in the project team.

The complete expression for maintenance effort is

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

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 $\Delta\text{SLOC}_k(i, i-1)$. 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 $\Delta\text{SLOC}_k(i, i-1)$.

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 $\text{ref}_k(i)$ which evaluates to 1 if a refactoring occurs, to 0 otherwise:

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

With this definition, $\text{ref}_k(i)$ 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^{ref} is defined as the ratio of the number of refactoring operations N_k^{ref} to the total number of versions V_k of application k:

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

Table 1: Types of operations on source code.

		$\Delta F_k(i, i-1)$		
		< 0	$= 0$	> 0
$\Delta \text{SLOC}_k(i, i-1)$	≤ 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^{ref} is defined as follows:

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

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 τ is defined as

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

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

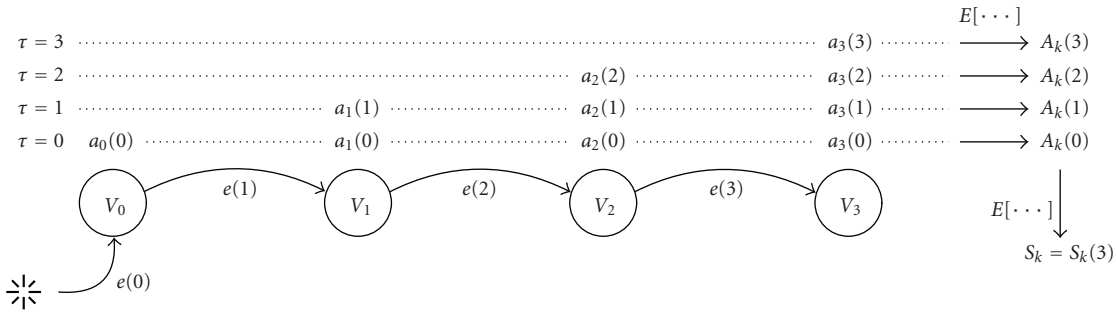

Figure 1: An example describing the computation of the time entropy S_k of a generic application with four versions.

different time lags. In Figure 1, $a_i(\tau)$ has been computed for all versions by considering all possible time lags, that is, τ ranging from 0 to 3. By calculating the mean value of all $a_i(\tau)$ for different values of τ (cf. Expression (15)), we calculate the average autocorrelation $A_k(\tau)$ of application k for each time lag τ. 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 τ) and on the older versions (high values of τ). Time entropy S_k is then calculated as a mean value over τ. 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 τ. When τ 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 α_k and β_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

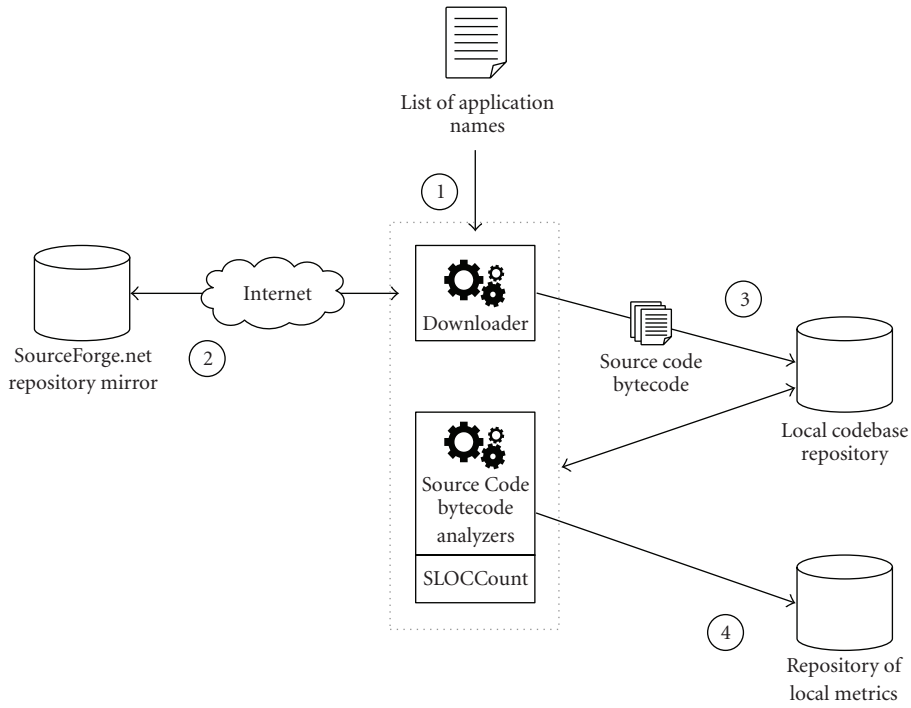

Figure 2: Action flow of the data analysis tool.

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 τ 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^{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^{ref}) versus time entropy variation (ΔS_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^{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.

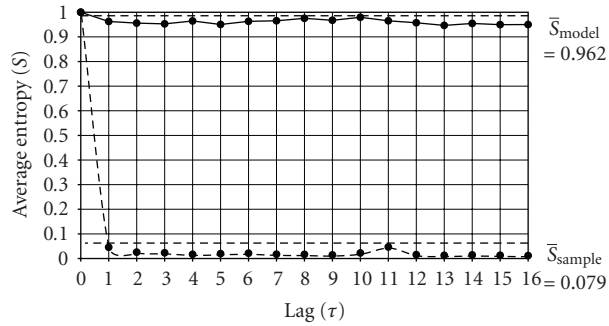

Figure 3: Time entropy of the application sample (dashed line, $\overline{S}_{sample}$) and reference application (continuous line, $\overline{S}_{model}$).

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	σ^2	$z_{1-\alpha}$	P-value	Result
(H1)	$\overline{S}_{model}$	0.962	$2\ 0E^{-4}$	2.58	$< .001$	Reject h_0
	$\overline{S}_{sample}$	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 (ΔS_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		ΔS_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 Refactoring Frequency	In community OS projects, time entropy cannot be related to periodic refactoring.
(H3)	Time Entropy 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

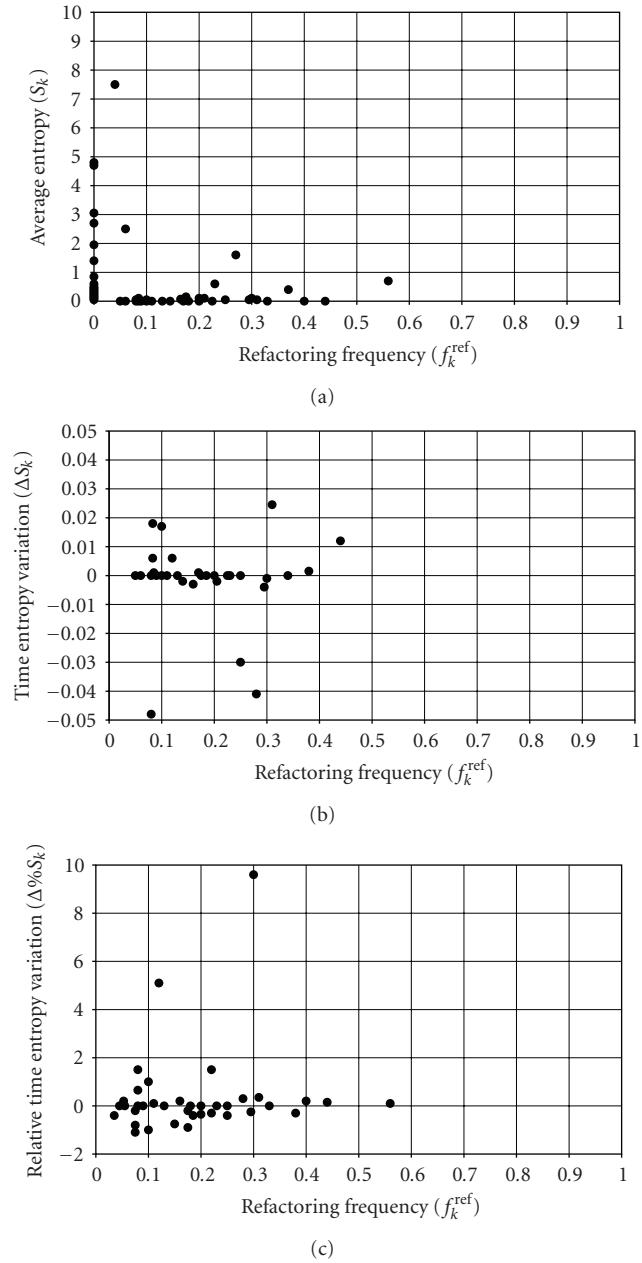

Figure 4: Scatter plots of refactoring frequency (f_k^{ref}) and (a) absolute average time entropy (S_k), (b) average time entropy variation after a refactoring operation (ΔS_k), and (c) average relative time entropy variation after a refactoring operation ($\Delta^{\%} S_k$). Each data point corresponds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

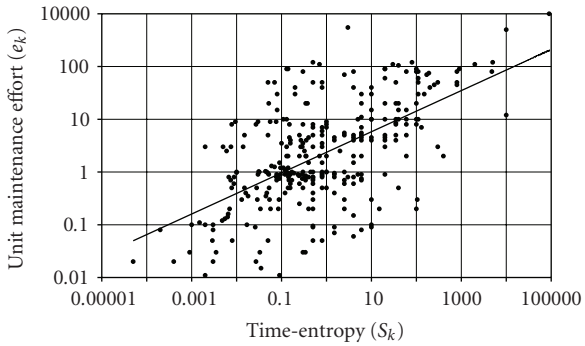

FIGURE 5: Scatter plot of time entropy (S_k) and average unit maintenance effort (e_k) variables. Axes in logarithmic scale, model $\overline{e}_k = 1.669 \cdot \overline{S}_k^{0.410}$, $R^2 = 0.461$, P value $\leq .001$.

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 ith observed data point and the ith 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.