2008

The Impact of Complexity on Software Design Quality and Costs: An Exploratory Empirical Analysis of Open Source Applications

Chiara Francalanci
Dipartimento di Elettronica e Informazione Politecnico di Milano, francala@elet.polimi.it

Francesco Merlo
Politecnico di Milano, merlo@elet.polimi.it

THE IMPACT OF COMPLEXITY ON SOFTWARE DESIGN QUALITY AND COSTS: AN EXPLORATORY EMPIRICAL ANALYSIS OF OPEN SOURCE APPLICATIONS

Francalanci, Chiara, Politecnico di Milano, Dipartimento di Elettronica e Informazione, via Ponzio 34/5, I-20133 Milano, Italy, francala@elet.polimi.it

Merlo, Francesco, Politecnico di Milano, Dipartimento di Elettronica e Informazione, via Ponzio 34/5, I-20133 Milano, Italy, merlo@elet.polimi.it

Abstract

It is well known that complexity affects software development and maintenance costs. In the Open Source context, the sharing of development and maintenance effort among developers is a fundamental tenet, which can be thought as a driver to reduce the impact of complexity on maintenance costs. However, complexity is a structural property of code, which is not quantitatively accounted for in traditional cost models.

This paper introduces the concept of functional complexity, which weights the well-established McCabe's cyclomatic complexity metric to the number of interactive functional elements that an application provides to users. Such metric is used to analyze how Open Source development costs are affected by complexity. Traditional cost models, like CoCoMo, do not take into account the impact of complexity in estimating costs by means of accurate indicators. In contrast, results show how a higher complexity is associated with a lower design quality of code, and, hence, higher maintenance costs. Consequently, results suggest that a reliable effort estimation should be based on a precise evaluation of software complexity. Analyses are based on quality, complexity, and maintenance effort data collected for 59 Open Source applications (corresponding to 906 versions) selected from the SourceForge.net repository.

Keywords: Open Source Software complexity, costs and quality.

1 INTRODUCTION

Authors in the software economics field concur that software maintenance accounts for the bulk of the costs over an application's life cycle (Lientz and Swanson 1981, Banker et al. 1993 and Boehm et al. 2004). The cost efficiency of maintenance interventions is affected by many factors: a fundamental cost driver is the complexity of code. Historically, many models of software cost estimation focused on the evaluation of development costs, without considering complexity metrics. In contrast, as suggested by a study on commercial software by Banker et al. (1993), high levels of software complexity account for approximately 25% of maintenance costs, which is equivalent to more than 17% of total life-cycle costs.

This paper addresses this issue by analyzing the impact of software functional complexity on design cost and quality metrics in an Open Source context. Since complexity is an inherent property of a software system, independent of the quality of design, this paper posits that a higher level of complexity negatively affects design quality and, consequently, development and maintenance costs, even if Open Source software matches high average quality standards (O'Reilly 1999 and Paulson et al. 2004). The paper discusses how this relationship between complexity and costs cannot be estimated with traditional cost models (such as CoCoMo), which do not account for complexity with objective measures.

The presentation is organized as follows. Section 2 presents an overview of the existing models for the evaluation of software quality, complexity, and costs. Section 3 presents the research hypotheses and testing method. Section 4 shows the statistical verification of research hypotheses, while Section 5 presents research results and future work.

2 RELATED WORK

The estimation of software development costs and the analysis of software properties by means of metrics have always been important research subjects in the software engineering community. The first and most referenced software metric is the number of source lines of code, commonly called SLOC, which is still broadly used (Park 1992). Despite, or even because of, the simplicity of this metric, it has been subjected to severe criticism (see Jones 1978, Rudolph 1983, Jones 1986, Canning 1986, Low and Jeffery 1990, Tegarden et al. 1992, Ghezzi et al. 2003). However, it can be considered a baseline metric used for comparison with other measures as suggested by Basili and Hutchens (1983). Apart from measures focusing on software size, complexity and quality represent the main focus of the research on software metrics. The research by McCabe (1976) and Halstead (1977) are considered cornerstone contributions in the field of complexity metrics, and have been widely used also in industrial contexts (see Halstead et al. 1976, Fitzsimmons and Love 1978 and Christensen et al. 1981).

The measurement of software quality is traditionally based upon complexity and design metrics. The first works proposed by Oviedo (1980) and Troy et al. (1981) based their evaluation on complexity metrics. More recently, the quality of software has been intended as a complex property, which can be studied from many perspectives. In particular, with the diffusion of the object oriented programming paradigm, the concept of quality has been tightly tied to the notion of coupling and cohesion (see Emerson 1984a, Emerson 1984b, Longworth et al. 1986). More recently, some metrics suites have been proposed to evaluate the quality of design of an object oriented software, like those by Chidamber and Kemerer (1994) and Brito e Abreu (1995); these works have been subject to a lively debate and in-depth analysis, which have proved the use of the metrics as indicators of the fault-proneness of software (see Basili et al. 1996, Rosenberg 1998, Chidamber et al. 1998, Briand et al. 1999 and Gyimothy et al. 2005).

Several models and techniques for cost estimation have been proposed (e.g., Basili et al. 1996a, Zhao et al. 2003, Boehm et al. 2004, Kemerer 1987 and Briand et al. 1999). The literature makes a distinction between the initial development cost and the cost of subsequent maintenance interventions. The latter has been empirically found to account for about 75% of the total development cost of an application over the entire application's life cycle (Lientz and Swanson 1981, Banker et al. 1993 and Boehm et al. 2004). The first and most widely used cost model is the Constructive Cost Model (CoCoMo), defined by Boehm in the early '80s (Boehm 1981). CoCoMo provides a framework to calculate initial development costs based on an estimate of the time and effort (man 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 Boehm et al. 1995). The latter takes into account requirements, in addition to SLOC. Requirements are measured in function points (FPs), i.e. the number of elementary operations performed by a software application (IFPUG 1999, Garmus and Herron 2001 and Ahn et al. 2003).

3 RESEARCH METHOD

This section presents our research approach. Section 3.1 discusses the research hypotheses, Section 3.2 presents the set of metrics that has been used for empirical analyses, Section 3.3 describes the data sample, while Section 3.4 reports data analyses and research results.

3.1 Research hypotheses

Software cost models are largely used both in industrial and academic contexts as a reference for estimating development and maintenance effort. These models, such as CoCoMo, are essentially size-based, since the fundamental parameter they rely on for effort estimation is the size of the software that has to be developed or maintained (see Section 2). Other factors, such as project complexity, platform-dependent characteristics or project reusability and reliability are taken into account only in a qualitative way. For example, in CoCoMo II the assessment of project complexity is performed through a subjective evaluation of the level of complexity, which has to be ranked on a six-degree scale, from *very low* to *extra high*. This lack of objective measures is a serious threat to the accuracy of the cost estimates provided by the models, since the same level of complexity might be perceived as more or less critical by different individuals (for example, depending on personal skills, experience, and even domain knowledge). Consequently, cost estimations related to software projects of comparable size could be similar even thought their complexity levels are actually different. However, as pointed out by Banker et al. (1993) and Banker and Slaughter (1996), complexity is a factor that actually influences software development and maintenance costs: the greater the complexity, the higher the unit development and maintenance costs. From these considerations follows the first research hypothesis:

H1: Traditional cost models fail to account for complexity. In contrast, complexity has a significant and non-negligible impact on costs.

As noted by Chan et al. (1996) and Kataoka et al. (2002), software quality can be improved through periodic refactoring interventions. Since the evidence from literature indicates that quality affects development and maintenance costs (Banker et al. 1993, Banker and Slaughter 1996), one might think that by restoring quality, refactorings can also reduce the complexity level and, hence, costs. However, as noted by Raymond (2004), each application has an inherent level of complexity that is determined by its requirements. A complex problem usually involves a complex solution. The inherent complexity of an application cannot be indefinitely reduced, since there is a threshold above which it is not possible to trade away features for simplicity. That is, the implementation of the functional

requirements of an application implies a minimum level of inherent complexity which cannot be eliminated by refactoring. This is summarized by the second research hypothesis:

H2: In an Open Source context, complexity and refactoring frequency are unrelated variables. Complexity is an inherent characteristic whose impact on costs cannot be mitigated by investing on quality.

However, a level of unnecessary complexity (cf. Raymond 2004) might be introduced if designers and programmers do not pay attention to good design and programming rules and practices, or simply do not follow the simplest way to implement a required set of features. Consequently, the level of complexity could be uselessly high, with a negative effect on development and maintenance costs. Open Source software is generally considered to meet high quality standards (cf. Fitzgerald 2004). The common perception is that the looser governance approach of open projects removes the pressure of deadlines and encourages the individual motivation of developers towards the production of a unique artefact (cf. Howison & Crowston 2005). As a consequence, we hypothesize that Open Source software is not affected by unnecessary complexity, and that its quality level is high. These considerations lead to our third research hypothesis:

H3: In an Open Source context, the average level of design quality is high with respect to literature benchmarks.

The literature indicates that applications with higher levels of complexity are harder to design and manage. Managing complexity involves an effort that may lead designers to focus on problem solving rather than code and design polishing. This leads to a negative effect of complexity on quality, as stated by our fourth research hypothesis:

H4: In an Open Source context, applications with greater complexity have a lower level of design quality.

3.2 Metrics of complexity and quality

The evaluation of software properties has been carried out through the measurement of a set of 18 metrics intended to asses various characteristics at different levels of granularity. Table 2 reports a detailed description of the complete metric set. Applications have been described from a general point of view through a set of "classic" metrics, intended to provide information about the size of the application (source lines of code, number of methods and number of interactive GUI functionalities). The inherent complexity of each application has been measured by introducing the concept of functional complexity, which is defined as follows. Given a system S of cyclomatic complexity CC_S, composed by its set of object classes $O = \{O_1, ..., O_n\}$ and given for each object class O_i its set of methods $M_i = \{M_i^1, ..., M_i^n\}$, we define the average functional complexity $FC(S)$ for system S as:

$$FC(S) = \frac{CC_S \cdot \sum_{O_i \in O} |M_i|}{FUN(S)},$$

where $FUN(S)$ is the number of functionalities of system S. The intuitive meaning of this metric is to assess how complex is an application given the number of interactive functionalities it provides to the users. To support the evaluation of applications' design properties, two of the most referenced suites of object-oriented design metrics have been included in the metrics set: in particular, the MOOD metric set (Brito e Abreu, 1995) for evaluations at system level, and the Chidamber and Kemerer metrics suite (Chidamber and Kemerer, 1994) for measurements at class level.

Maintenance effort has been estimated by measuring the days elapsed between the release of subsequent versions, weighted by the number of active developers of each project. Considering the i-th version v_k^i of application k, the maintenance effort for the subsequent version j is:

$$ME(j) = [date(v_k^j) - date(v_k^i)] \cdot a_k \cdot EAF,$$

where a_k is the number of active developers of application k (for the purposes of this study, a_k has been set equal to the number of project administrators as indicated on each project home page on SourceForge.net). *EAF* is an effort adjustment factor which has been estimated with an empirical survey of 940 active developers of the SourceForge.net community: 87 contributors have responded to the survey, allowing the estimation of the effort adjustment factor for about 50% of the projects in our sample. Average values have been used for the rest of the projects. The survey was aimed at assessing the fraction of time that each developer spends in development activities related to the Open Source project he or she is involved in. The adjustment factor allows us to address the fact that Open source developers may not work full time and that they might be involved in more than one project at a time. Table 1 presents some statistics from the survey. Confidence intervals (C.I.) at a significance level of $\alpha = 0.05$ have been computed to verify the statistical significance of the surveyed mean values.

Variable	**n**	**Avg**	**St.Dev.**	**C.I.**
Hr/Week (Admin)	29	8.09	9.65	±4.24
Hr/Week (Devel)	56	8.62	12.55	±3.86

Table 1 Development effort of Open Source developers.

Maintenance costs have been estimated using the CoCoMo model (Boehm 1981, Boehm 2000) in its basic form. Model parameters have been weighted for the evaluation of organic projects, which are defined as relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements. This seems to be an appropriate description of the projects selected for this study. The expression of the cost model used for the study is:

$$MC = a \cdot KSLOC^b,$$

where *MC* is the estimated maintenance cost (expressed in man months), *KSLOC* is the size of the application to be developed and a and b are parameters whose values are $a = 2.40$ and $b = 1.05$.

Metric	**Description**
SLOC (Source lines of Code)	Physical non white and non comment lines of code
CC (Cyclomatic Complexity)	Decision points in the program flow
FUN (Functionalities)	Interactive GUI elements
M (Methods)	Methods of the application
ME (Maintenance Effort)	Empirical estimate
MC (Maintenance Cost)	CoCoMo estimate
WMC	Weighted method complexity for a class
DIT	Max path length from a class to root of inheritance tree
RFC	Potentially executable methods in response to a received message
NOC	Immediate subclasses of a class
CBO	Distinct classes used by a given class (excludes inheritance relationships)
LCOM	Average % of methods accessing different attributes in a class
COF	Average number of coupling relationships
AHF	Percentage of hidden attributes
MHF	Percentage of hidden methods
AIF	Percentage of inherited attributes
MIF	Percentage of inherited methods
PF	Actual polymorphic situations

Table 2 Metric set for software evaluation

3.3 Data Sample

The data set used for this study has been directly measured on the source code of a sample of Open Source community applications taken from the SourceForge.net repository. The data sample has been

selected to preserve the heterogeneity of the classification domains of SourceForge.net, by considering a significant subset of applications. Since mining on line repositories such as SourceForge.net can lead to controversial results because of the varying quality of available data (Howison and Crowston 2004), applications have been selected according to the following criteria:

- Project Maturity: beta status or higher. Less mature applications have been excluded because of their instability and low significance.
- Version History: at least 5 versions released.
- Domain: selected applications are uniformly distributed across the SourceForge.net domain hierarchy.

The initial set of applications has been automatically cleaned by removing all the applications that were not consistent with our quality requirements, leading to dataset DS_1.

3.4 Data Analyses

The goal of our analyses is to investigate how design quality metrics of Open Source software are affected by the complexity of source code. This goal is achieved by analyzing the temporal variations of the data describing our application sample. Dataset DS_1 has been manually preprocessed to identify and correct systematic errors, such as invalid or non significant application versions downloaded and incorrectly included as part of the sample. Preprocessing has been focused on time-dependent variables, such as the number of source lines of code (*SLOC*), functionalities (*F*), methods (*M*) and related variations ($\Delta SLOC$, ΔF and ΔM).

First, variables have been analyzed visually to identify outliers. Results have been used as a starting point for more accurate analyses. Time series typically have a structure, such as auto-regression or trend, that should be identifiable. Autoregressive linear analysis was performed to check for auto-regression. A first order AR(1) model has been solved for each application in the sample by considering unit maintenance costs in addition to the time-dependent variables described above. The analysis of the average deviation computed by means of the AR(1) models led to the identification of 5 applications which showed high deviation values for all the variables. The last step has been the analysis of trend curves, which has been performed by analyzing data series fitting with linear, quadratic, and logistic growth curves for the variables *F* (functionalities), *M* (methods) and *SLOC* (source lines of code) of each application.

These preprocessing analyses led to the identification of a subset of 18 applications (about 20% of the sample size) characterized by potential anomalies. For each one of these applications, a manual inspection of data was performed to identify the cause of inconsistencies. This refinement step lead to the identification of the following error classes, reported together with distribution percentages in DS_2:

- *Repeated versions (13.3%)*: Application archives contained the same version, stored with different file formats (such as .zip or .tar.gz).
- *Unrelated data archives (20.0%)*: Application archives contained patches or plugins, that should not be considered as versions.
- *Damaged data archives (2.7%)*: Application archives could not be analyzed because of compression errors.
- *Unavailable libraries (8.0%)*: Application used libraries which could not be retrieved.
- *Static analyzer problems (56.0%)*: Internal problems of the static analysis library engine with respect to the application source code.

The final data sample, DS_2, is composed by 59 applications, corresponding to 906 versions. Table 3 presents the summary statistics of the DS_2 dataset.

Variable	Average	Median	Min	Max	St.Dev.
Active Developers	1.53	1	1	9	1.25
versions	15.35	8	2	226	13.49
Mean Size (SLOC)	13311	6910	905	61184	14173

Table 3 Descriptive statistics for DS_2.

Source code has been analyzed with a tool developed *ad-hoc*. The tool provides data on all the metrics described in Section 3.2, performing static analyses of Java source code. The static analysis engine is based on the Spoon compiler (Pawlak 2005), which provides the user a representation of the Java abstract syntax tree in a metamodel that can be used for program processing. The perspective adopted by the tool is explicitly evolutionary: given an application, metrics measurements are carried out on each version, by computing not only the values but also the variations between subsequent versions.

4 EXPERIMENTAL RESULTS

This section presents the empirical findings of the study, by describing how research hypotheses have been verified by means of statistical tests.

Hypothesis H1 is verified by showing that CoCoMo estimates of maintenance effort (*MC*) do not show significant differences in applications with high functional complexity with respect to applications with low levels of complexity, while our empirical estimates (*ME*) highlight such differences. To achieve this, the first step is to split the data sample into two clusters on the basis of functional complexity values. The two clusters, namely FC_{LO} and FC_{HI} , have been created by dividing applications with functional complexity below (FC_{LO}) and above (FC_{HI}) the median value. Table 4 presents the summary statistics and properties of the two clusters.

Variable	Cluster FC_{LO}	Cluster FC_{HI}
Cardinality	29	30
Versions	18.48	12.33
SF.net Ranking	10410.74	10192.79
Downloads	51918.48	114856.13
Months	25.79	31.03
Months/Version	2.95	3.58
Refactoring Freq.	0.22	0.24
SLOC	11312.97	15244.37
F	112.06	52.65
M	862.40	1350.73
CC	2.50	3.04
Maintenance Effort (ME)	439.38	556.47
CoCoMo Cost Estimate (MC)	0.014	0.015

Table 4 Cluster properties.

Hypothesis H1 is verified if the following two conditions hold:

- C1: The average CoCoMo maintenance effort estimate MC_{HI} of applications in cluster FC_{HI} is not significantly different from the average MC_{LO} of applications in cluster FC_{LO};
- C2: The average empirical maintenance effort estimate ME_{HI} of applications in cluster FC_{HI} is greater than the average ME_{LO} of applications in cluster FC_{LO}.

A one-way ANOVA test has been set up on the following null and alternative hypotheses:

C1: h_0: $E[MC_{LO}] = E[MC_{HI}]$; h_1: $E[MC_{LO}] \neq E[MC_{HI}]$

C2: h_0: $E[ME_{LO}] = E[ME_{HI}]$; h_1: $E[ME_{LO}] \neq E[ME_{HI}]$

Table 5 presents the results of the tests. As shown in Table 5, the null hypothesis of condition C1 cannot be rejected (significance level is .653), while the null hypothesis of condition C2 must be rejected. Thus, hypothesis H1 is verified, since no significant differences can be found between CoCoMo estimates (C1), while empirical estimates reveal a significant difference of maintenance effort between the two clusters of applications (C2).

Condition	Variable	n	Average	H0	F	Sig. (2-tail)
C1	$E[MC_{LO}]$	29	0.014	cannot reject	0.204	.653
	$E[MC_{HI}]$	30	0.015			
C2	$E[ME_{LO}]$	527	439.38	reject	16.958	.000
	$E[ME_{HI}]$	370	556.47			

Table 5 One-way ANOVA test for H1.

Hypothesis H2 has been verified by testing various types of correlations between refactoring frequency and functional complexity. Refactoring frequency has been operationalized according to Capra et al. (2006). Data series of refactoring frequency and functional complexity values have been studied by means of a regression analysis. The evaluation has been performed through the analysis of the R^2 coefficient of determination for each kind of relation, which states 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 functional complexity values, since R^2 values are too low for all the considered cases. Figure 1 presents a scatter plot of refactoring frequency and corresponding entropy values. These results support H2, confirming that in an Open Source context the functional complexity of a system is not correlated with the frequency of refactoring interventions.

Relation	R^2	Equation
Linear	.0017	$y = -0.8x + 2.9$
Quadratic	.0020	$y = -2.0x^2 + 0.5x + 2.8$
Cubic	.0247	$y = 60.3x^3 - 64.5x^2 + 17.2x + 1.7$
Exp	.0024	$y = 2.6e^{-0.2x}$
Log	$.8 \cdot 10^{-7}$	$y = -0.003ln(x) + 2.8$
Power	.0003	$y = 2.4x^{-0.01}$

Table 6 Regression analyses for H2.

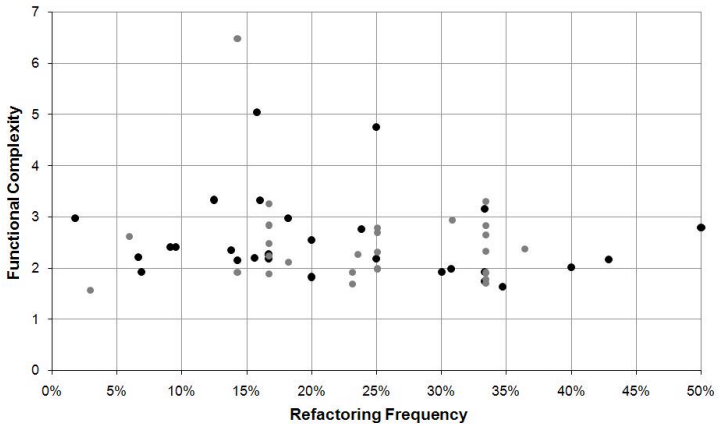

Figure 1 Scatter plot for H2.Black points are applications belonging to cluster FC_{LO}, while grey points are applications belonging to cluster FC_{HI}.

Hypothesis H3 is verified by comparing the values of design quality metrics of the applications in the data sample to literature benchmarks. A whole body of literature focuses on validating software metric suites, by providing theoretical and mathematical proofs of validity for each proposed metric (see, for example, Basili 1996b, Harrison et al. 1998 and Gyimothy et al. 2005). However, a few works provide reference empirical values. We refer to the work by Laing and Coleman (2001), which presents the results of a quality assessment project at SATC (NASA). They report the metrics of Chidamber and Kemerer Suite (Chidamber and Kemerer 1994), which have been evaluated for three different software systems. Since for each system an aggregated quality index is available from SATC full-scale code analysis, related metrics values can be considered as significant benchmarks. Table 7 provides a comprehensive view of metrics values for data sample DS_2 and for the three reference systems analyzed by Laing and Coleman (2001), which we identify by means of the corresponding aggregate quality index (*Low*, *Medium* or *High*). As can be noted from the table, the average values of four out of six metrics (namely, LCOM, RFC, NOC and WMC) are higher than benchmark values, while only two metrics out of six (namely, CBO and DIT) are lower than benchmark values. As a consequence, the overall quality level of data sample DS_2 can be assumed as good, at least with respect to the SATC quality levels.

	Sample			**Benchmark (Low)**			**Benchmark (Med.)**			**Benchmark (High)**		
	Avg	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
CBO	2.48	0.29	5	2.48	0	11	2.09	0	16	1.25	0	22
LCOM	37.89	0.38	619	447.65	0	3804	113.94	0	3281	78.34	0	5444
RFC	12.74	0.88	52	80.39	0	381	28.60	0	457	43.84	0	827
NOC	0.19	0.00	1	0.07	0	2	0.39	0	116	0.35	0	21
DIT	0.24	0	1	0.37	0	2	1.02	0	6	0.97	0	4
WMC	18.59	4.55	134	45.70	0	596	23.97	0	492	11.10	0	381

Table 7 Benchmarks for H3.

Hypothesis H4 is verified by comparing the values of design quality metrics of the two application clusters FC_{LO} and FC_{HI}, in order to test whether applications in cluster FC_{LO} show a better design quality level with respect to the applications of cluster FC_{HI}. Before comparing the average cluster values for each metric, a *t*-test has been performed in order to verify the statistical significance of the difference between the average values themselves. Table 8 presents the results of the *t*-test. The hypothesis stating that metrics averages are different between the two application clusters can be proved for all metrics with a significance value lower than .001. The only exception is metric MHF, showing a difference between the two cluster average values that cannot be considered statistically significant, since the significance value is .237 (higher than .05).

Metric	**T**	**df**	**Sig. (2-tailed)**	**Mean diff.**	**Std. Err. diff**	**95% Conf. Int.**	
						Low	High
NOC	-9,93	611	1,24E-21	-0,523	0,053	-0,627	-0,420
CBO	-11,93	517	3,94E-29	-0,163	0,014	-0,190	-0,136
RFC	-11,40	832	4,51E-28	-0,881	0,077	-1,033	-0,729
LCOM	-14,44	592	9,93E-41	-0,132	0,009	-0,149	-0,114
DIT	-12,81	637	1,43E-33	-4,005	0,313	-4,619	-3,391
WMC	-8,09	424	6,16E-15	-8,292	1,024	-10,306	-6,278
AHF	-9,88	878	6,51E-22	-0,118	0,012	-0,142	-0,095
MHF	-9,42	642	7,79E-20	-0,080	0,009	-0,097	-0,064
AIF	-3,26	532	1,21E-03	-0,012	0,004	-0,019	-0,005
MIF	-1,18	901	2,37E-01	-0,009	0,008	-0,024	0,006
COF	-3,89	657	1,13E-04	-0,047	0,012	-0,071	-0,023
PF	-6,12	652	1,65E-09	-0,073	0,012	-0,096	-0,049

Table 8 t-test results for H4.

Table 9 presents the comparison between the average cluster values of each design quality metric. The percent difference between clusters FC_{LO} and FC_{HI}, is also reported. As it can be noted, seven out of twelve metrics (namely, CBO, DIT, LCOM, NOC, RFC, WMC and COF) show a better design quality level in cluster FC_{LO}, thus confirming hypothesis H4.

Metric	FC_{LO}	FC_{HI}	% Increase	Comparison
CBO	2,026	2,576	27% **	better
DIT	0,185	0,303	64% **	better
LCOM	29,252	57,322	96% **	better
NOC	0,152	0,240	58% **	better
RFC	10,725	14,656	37% **	better
WMC	16,267	24,038	48% **	better
AHF	0,661	0,743	12% **	worse
AIF	0,078	0,128	63% **	worse
COF	0,036	0,053	50% **	better
MHF	0,245	0,222	-10%	n.s.
MIF	0,171	0,238	39% **	worse
PF	0,143	0,169	18% **	worse
** *denotes significance at 0.01 level (2-tailed).*				

Table 9 Comparison of cluster average design quality metrics.

5 DISCUSSION AND CONCLUSIONS

Results indicate that complexity is an inherent property of source code, and can be hardly influenced by interventions oriented at restoring quality, such as refactorings. This is motivated by the fact that software's inherent complexity is strictly tied to the fulfilment of functional requirements, and cannot be reduced or simplified beyond a certain threshold. Moreover, results suggest that traditional cost models such as CoCoMo fail in accounting for the impact of complexity on development and maintenance costs. Complexity should be accounted for, since it generally implies lower design quality as posited and proved by hypothesis H4. Hence, a reliable cost estimation must be based on cost models that do not neglect complexity. These considerations are strengthened by the fact that the empirical evidence on maintenance effort indicates that the average effort for applications with higher levels of functional complexity is 27% higher with respect to the average effort for applications with lower complexity levels. It is worth noting that the effect of complexity is not accounted for by CoCoMo estimates, which do not show any effort difference between the two clusters. Although these considerations are based on a preliminary empirical analysis, some valuable results have been obtained. Given that complexity considerably affects other design quality metrics (and, hence, development and maintenance costs) even in a fine software quality context like Open Source, a precise evaluation of complexity should not be neglected, since its impact is likely to be substantial.

5.1 Future work

Future work is focused on defining the causal relationships among quality metrics by means of regression analyses. This will allow us to better describe which are the driving quality dimensions affecting maintenance costs. Another issue our future work will address is the enlargement of the application sample, in order to provide a better data base for improving the statistical significance of hypotheses testing.

ACKNOWLEDGMENTS

We would like to thank Mauro Colombo for his aid in the data collection and analysis phases.

References

Ahn, Y., Suh, J., Kim, S., & Kim, H. (2003). A software maintenance project effort estimation model based on function points. Jnl. Software Maintenance and Evolution: Research and Practice, 15 (2), 71–85.

Banker, R., Datar, S., Kemerer, C., & Zweig, D. (1993). Software complexity and maintenance costs. Comm. ACM, 36 (11), 81–94.

Banker, R. D. & Slaughter, S. A. (1996). A study on the effects of software development practices on software development effort. In Proc. Int'l Conf. Software Maintenance.

Basili, V., Briand, L., Condon, S., Kim, Y. M., Melo, W. L., & Valett, J. D. (1996a). Understanding and predicting the process of software maintenance releases. Proc. Int'l Conf. Software Eng., 464–474.

Basili, V. & Hutchens, D. (1983). An empirical study of a complexity family. IEEE Trans. Software Eng., 9 (6), 664–672.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996b). A validation of object-oriented design metrics as quality indicators. IEEE Trans. Software Eng., 22, 751–761.

Boehm, B. (1981). Software Engineering Economics. Prentice-Hall, NJ.

Boehm, B. (2000). Software cost estimation with COCOMO II. Prentice Hall, NJ.

Boehm, B., Brown, A. W., Madacy, R., & Yang, Y. (2004). A software product line life cycle cost estimation model. Proc. Int'l Symposium on Empirical Software Eng.,156–164.

Boehm, B., Clark, B., Horowitz, E., Madachy, R., Shelby, R., & Westland, C. (1995). The COCOMO II software cost estimation model. Int'l Society of Parametric Analysts.

Briand, L. C., El Emam, K., Surmann, D., Wiezczorek, I., & Maxwell, K. D. (1999b). An assessment and comparison of common software cost estimation modelling techniques. Proc. Int'l Conf. Software Eng., 313–323.

Brito e Abreu, F. (1995). The MOOD metrics set. In Proc. of ECOOP Workshop on Metrics.

Canning, R. G. (1986). A programmer productivity controversy. EDP Analyser, 24 (1), 1–11.

Capra, E., Francalanci, C., Merlo, F. & Tosetti, M. (2006) Cost implications of Open Source Software Commonality and Reuse. In Proc. Europ. Conf. Information Systems.

Chan, T., Chung, S., & Ho, T. (1996). An economic model to estimate software rewriting and replacement times. IEEE Trans. Software Eng., 22 (8), 580–598.

Chidamber, S. & Kemerer, C. (1994). A metrics suite for object oriented design. IEEE Trans. Software Eng., 20, 476–493.

Chidamber, S. R., Darcy, D. P., & Kemerer, C. F. (1998). Managerial use of metrics for object-oriented software: An exploratory analysis. IEEE Trans. Software Eng., 24, 629–639.

Christensen, K., Fistos, G. P., & Smith, C. P. (1981). A perspective on software science. IBM Systems Jnl., 20 (4), 372–387.

Emerson, T. J. (1984a). A discriminant metric for module comprehension. In Proc. Int'l Conf. Software Eng., 294–431.

Emerson, T. J. (1984b). Program testing, path coverage and the cohesion metric. In Proc. of COMPSAC84, 421–431.

Fitzgerald B. (2004). A critical look at Open Source. IEEE Computer, 37 (7), 92-94.

Fitzsimmons, A. & Love, T. (1978). A review and evaluation of software science. ACM Computing Surveys, 10 (1), 3–18.

Garmus, D. & Herron, D. (2001). Function Point Analysis. Addison Wesley.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2003). Fundamentals of Software Engineering. Prentice Hall.

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Software Eng., 31, 897–910.
Halstead, M. H. (1977). Elements of Software Science. Elsevier Computer Science Library.
Halstead, M. H., Gordon, R. D., & Elshoff, J. L. (1976). On software physics and GM's PL-I programs. Tech. Rep. GMR-2175, General Motors Research Laboratories, Warren, MI.
Harrison, R., Counsell, S. J., & Nithi, R. V. (1998). An evaluation of the MOOD set of object-oriented software metrics. IEEE Trans. Software Eng., 24 (6), 491–496.
Howison, J. & Crowston, K. (2004). The perils and pitfalls of mining SourceForge. In Proc. Int'l Workshop on Mining Software Repositories, 7–12.
Howison, J. & Crowston, K. (2005). The social structure of free and open source software development. First Monday, 10 (2).
IFPUG (1999). Function Point Computing Practices Manual, Release 4.1. IFPUG, Westerwille, Ohio.
Jones, T. C. (1978). Measuring programming quality and productivity. IBM Systems Jnl., 17, 39–63.
Jones, T. C. (1986). Programming productivity. McGraw-Hill.
Kataoka, Y., Imai, T., Andou, H., & Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. Proc. Int'l Conf. Software Maintenance, 576–585.
Kemerer, C. F. (1987). An empirical validation of software cost estimation models. Comm. ACM, 30 (5), 416–430.
Laing, V. & Coleman, C. (2001). Principal component of orthogonal object-oriented metrics. Tech. Rep. 323-08-14, Software Assurance Quality Centre, NASA.
Lientz, B. & Swanson, B. (1981). Software Maintenance Management. Addison-Wesley.
Longworth, H. D., Ottenstein, L. M., & Smith, M. R. (1986). The relationship between program complexity and slice complexity during debugging tasks. In Proc. of COMPSAC86.
Low, G. C. & Jeffery, D. R. (1990). Function points in the estimation end evaluation of the software process. IEEE Trans. Software Eng., 16, 16.
McCabe, T. J. (1976). A complexity measure. In Proc. Int'l Conf. Software Eng., 407.
OReilly, T. (1999). Lessons from open-source software development. Comm. ACM, 42 (4), 32–37.
Oviedo, E. I. (1980). Control flow, data flow and programmers complexity. In Proc. of COMPSAC80. 146–152.
Park, R. E. (1992). Software size measurement: a framework for counting source statements. Tech. Rep. SEI-92-TR-020, Software Eng. Institute, Pittsburg.
Paulson, J. W., Succi, G., & Eberlein, A. (2004). An empirical study of open-source and closed-source software products. IEEE Trans. Software Eng., 30 (4), 246–256.
Pawlak, R. (2005). Spoon: Annotation-driven program transformation - the AOP case. In Proc. Workshop on Aspect-Orientation for Middleware Development.
Raymond, E. S. (2004). The Art of Unix Programming. Addison Wesley.
Rosenberg, L. H. (1998). Applying and interpreting object oriented metrics. Tech. rep., Software Assurance Technology Centre NASA SATC.
Rudolph, E. E. (1983). Productivity in computer application development. Tech. rep., University of Auckland, Department of Management Studies, Australia.
Tegarden, D., Sheetz, S., & Monarchi, D. (1992). Effectiveness of traditional software metrics for object-oriented systems. In Proc. Hawaii Int'l Conf. System Sciences.
Troy, D. & Zweben, S. (1981). Measuring the quality of structured design. Jnl. System and Software, 2, 113–120.
Zhao, Y., Kuan Tan, H. B., & Zhang, W. (2003). Software cost estimation through conceptual requirement. Proc. Int'l Conf. Quality Software, 141–144.