Longitudinal Evaluation of Software Quality Metrics in Open-Source Applications

Arthur-Jozsef Molnar, Alexandra Neamţu and Simona Motogna
Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Keywords: Software Product Quality, Software Metrics, Metric Correlations, Longitudinal Case Study.

Abstract: Assessment of software quality remains the focus of important research efforts, with several proposed quality models and assessment methodologies. ISO 25010 describes software quality in terms of characteristics such as reliability, security or maintainability. In turn, these characteristics can be evaluated in terms of software metric values, establishing a relation between software metrics and quality. However, a general metric-based model for software quality does not yet exist. The diversity of software applications, metric definitions and differences between proposed quality models all contribute to this. Our paper proposes a longitudinal evaluation of the metric values and their relations in the context of three complex, open-source applications. We cover the entire 18 year development history of the targeted applications. We explore typical values for metrics associated with software product quality and explore their evolution in the context of software development. We identify dependant metrics and explore the effect class size has on the strength of dependencies. At each step, we compare the obtained results with relevant related work in order to contribute to a growing pool of evidence towards our goal - a metric-based evaluation of software quality characteristics.

1 INTRODUCTION

Software development has undergone large changes in the past decades. Technological advancement, together with large-scale digitization have lead to increasing customer expectations regarding functionality, ease of use and security. Software applications are increasingly complex. Methodologies such as Agile were developed to provide timely response to changes in requirements, as well as to deliver working software faster.

Increased system complexity requires new methodologies and tools to ensure that software remains reliable, maintainable and secure. Given that *"you cannot control that which you cannot measure"* (DeMarco, 1986), it stands that software measurement plays an important role in assessing software quality characteristics. The field of software metrics has kept pace with the rapid development of programming languages and paradigms, with many proposed metrics that can play a role in assessing software products.

Standards and methodologies for management and quality assurance, such as those of the ISO family[1] depend on quantitative data in order to measure key aspects of software product quality. Software metrics provide such data, and there exists a large body of research illustrating the relation between software metric values and product quality (Chidamber and Kemerer, 1994; Marinescu, 2005; Xu et al., 2008; Kanellopoulos et al., 2010; Molnar and Motogna, 2017).

However, many of the studies that explore the relation between software metrics and product quality acknowledge that more data is required before general models can be developed (Barkmann et al., 2009). This is especially true given research that evaluates metric data in a new light (Landman et al., 2014), or that which challenges generally held assumptions (Emam et al., 2001).

Our paper's main objective is to provide a sound foundation that can be leveraged in studying the relation between software metric values and product quality attributes. To achieve this, we propose a study that complements and extends existing research on one hand, and which fills identified gaps in the existing body of knowledge. Our paper's contributions can be summed up as follows:

(i) A quantitative evaluation of metric values for all released versions of three complex, open-source software applications.

[1] ISO 9126 and the newer ISO 25010 more specifically

Molnar, A., Neamţu, A. and Motogna, S.
Longitudinal Evaluation of Software Quality Metrics in Open-Source Applications.
DOI: 10.5220/0007725600800091
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 80-91
ISBN: 978-989-758-375-9

(ii) A long-term exploratory study on the evolution of metric values associated with software product quality. Our study evaluates the entire public development history of the target applications over the course of 18 years.

(iii) Identification of statistically correlated method pairs. We evaluate correlation strength in the context of each application together with studying the confounding effect of class size on dependent metrics.

(iv) An evaluation of our results in the context of previous research that employed similar methodology and compatible software tooling. Most of the related work consists of singular efforts difficult to incorporate in future work due to unclear study methodology, software tooling or differences between metrics.

A first important difference between the existing literature and our proposed study lays in the selection of target applications. The prevalent approach is to either hand-pick a number of applications for which several versions are studied (Kanellopoulos et al., 2010; Silva and Costa, 2015), or to carry out a cross-sectional study of a large number of applications (Barkmann et al., 2009; Landman et al., 2014; Lenhard et al., 2018). Our study includes the initial application versions, which are rudimentary with regards to functionality and bug-prone. We also study the most current implementations, that have an expanded feature set and enjoy a large user base. This allows us to study how the values and relations between studied metrics evolve during long-term software development.

A second important difference regards the selection of software metrics and extraction tools. We selected metrics that express the most important characteristics of object-oriented software: complexity, inheritance, coupling and cohesion (Marinescu, 2005; ARISA Compendium - Understandability for Reuse, 2018) and that were also employed in previous research targeting software quality (Landman et al., 2014; Silva and Costa, 2015; Lenhard et al., 2018).

Given that for most metrics there exist several definitions (Lincke et al., 2008; Bakar and Boughton, 2012), the choice of metric tool is crucial for obtaining comparable results. This is compounded by the fact that existing papers based on metric values often omit to specify the software tooling employed for value extraction. To address this, we've used the VizzMaintenance tool[2], which provides clear definitions for the metrics. We discuss these in the following section. More so, our approach allows us to directly compare our results with (Barkmann et al., 2009), where authors undertake a cross-sectional study of 146 software projects that are sanitized in a manner similar with ours and where the same tooling is used to extract metric information. To the best of our knowledge, (Barkmann et al., 2009) is the most relevant paper that we can use to validate and further explore our findings.

2 PRELIMINARIES

2.1 Software Metrics

Software metrics are functions used to measure some property of source code or its specification. The number and type of metrics available closely follows software development trends. Some of the earliest defined metrics are some of the most widely used and measure the number of source code lines, functions, or modules in a system. Paradigms such as object-orientedness resulted in new metrics, suited to describe their specific concepts. In the case of object-oriented software, these are structural complexity, coupling, cohesion and inheritance (Marinescu, 2005). Perhaps the most well-known suite of such metrics is the Chidamber & Kemerer (CK) metric suite (Chidamber and Kemerer, 1994).

Values for most established metrics can be extracted using metric extraction tools. These are available both as plugins, such as Metrics2 [3] for Eclipse, MetricsReloaded[4] for IntelliJ, or standalone tools such as JHawk[5]. When doing so, one must take into account the exact definition employed by the extraction tool. Considering comment or empty lines changes the reported number of lines of code. Counting inherited attributes and methods affects the numbers reported for a class. The lack of cohesion in methods (LCOM) metric, first defined by CK in (Chidamber and Kemerer, 1994), was refined by Li and Henry (Li and Henry, 1993a), and then by Hitz and Montazeri (Hitz and Montazeri, 1995). Studies such as (Lincke et al., 2008; Bakar and Boughton, 2012) explore these differences as well as their impact when building metric-based software quality models.

The situation is further confounded by differences between programming languages. Basili et al. (Basili et al., 1996) find that C++ features such as multiple inheritance, templates or friends are not covered by

[2]http://www.arisa.se/vizz_analyzer.php

[3]http://metrics.sourceforge.net

[4]https://plugins.jetbrains.com/plugin/93-metricsreloaded

[5]http://www.virtualmachinery.com/jhawkprod.htm

CK metrics. (Xu et al., 2008) and (Succi et al., 2005) show that apparent relations between metric values and software quality are language dependent.

As such, this section details the metric definitions that were used in our research. Our selection includes established size metrics together with some of the most studied object-oriented metrics, including the CK suite. All reported values were extracted using VizzAnalyzer. Formal definitions for calculated metrics are available at (ARISA Compendium - Understandability for Reuse, 2018). The tool was previously used in (Barkmann et al., 2009; Lincke et al., 2008), making our reported results directly comparable.

- *Coupling Between Objects (CBO)* - number of classes coupled to the one being measured. Two classes are said to be coupled when one of them uses methods or variables declared in the other one. CBO is a cohesion metric that is indicative of the effort needed for maintaining a class, as well as testing it (Rodriguez and Harrison, 2001).
- *Data Abstraction Coupling (DAC)* - measures the coupling complexity by counting the number of referenced abstract data types. The measurement excludes classes from the Java platform.
- *Depth of Inheritance Tree (DIT)* - measures the maximum length of a path from the class node to the inheritance hierarchy's root. This definition also covers the possibility of C++ like multiple inheritance. DIT is a structural metric defined for object-oriented systems (Rodriguez and Harrison, 2001).
- *Lack of Cohesion in Methods (LCOM)* - the number of method pairs which have no cohesion via common instance variables minus the number of method pairs which have (Rodriguez and Harrison, 2001).
- *Improvement to Lack of Cohesion in Methods (ILCOM)* - employs Hitz and Montazeri's updated definition for the LCOM metric (Hitz and Montazeri, 1995).
- *Locality of Data (LD)* - ratio between the local and all the variables accessed by the class. LD is a coupling metric defined in (Hitz and Montazeri, 1995). Local data includes non-public and inherited attributes, together with attributes accessed via getters. Existing research (ARISA Compendium - Understandability for Reuse, 2018) posits a relation exists between LD and potential for reuse and testability.
- *Lines of Code (LOC)* - number of lines of code for the given class. While LOC belongs to the first software metrics, it remains applicable among the widest varieties of programming languages and retains utility when investigating the quality of a software product (Rodriguez and Harrison, 2001). Furthermore, studying the relation between LOC and object-oriented metrics provides information about the relation between system size and structure. Furthermore, existing research (Emam et al., 2001) shows LOC to have a confounding effect that must be controlled when building metric-based quality models.
- *Message Pass Coupling (MPC)* - the number of method calls to methods defined in other classes. MPC is a coupling metric illustrating the dependency on other system classes.
- *Number of Attributes and Methods (NAM)* - size metric that provides the number of static and instance attributes and methods defined by the class. Does not count constructors, inherited fields or attributes.
- *Number of Children (NOC)* - measures the number of classes that inherit from the given class. Together with DIT, NOC is an inheritance related metric (Rodriguez and Harrison, 2001; Sarker, 2005).
- *Number of Methods (NOM)* - measures the number of methods locally defined by a class. Does not count inherited methods, or constructors. The number of locally added attributes for a class can be computed as $NAM - NOM$.
- *Response For a Class (RFC)* - measures the number of methods that can be invoked in response to a message of an object of a certain class or of some method from that class. This metric counts the number of calls to other classes from a certain one (Rodriguez and Harrison, 2001). RFC is a complexity metric.
- *Tight Class Cohesion (TCC)* - measures the cohesion between the public methods of a class. Defined as the ratio between the number of public method pairs that use the same instance of a class attribute divided by the total number of public method pairs of the class (Ott et al., 1970).
- *Weighted Method Count (WMC)* - measures the complexity of a class, using the McCabe cyclomatic complexity to weight methods. Like RFC, WMC is a complexity metric which provides an indication of the effort required to maintain the class (Rodriguez and Harrison, 2001).

2.2 Software Quality Models

One of the first models for software quality was the McCall model developed in 1976. It consisted in 55 quality characteristics, or factors, grouped in 11 main characteristics. The Boehm model added new factors to the McCall model and emphasized the maintainability aspect. Several others were created, including the Dromey and FURPS models (Bassam Al-Badareen et al., 2011; Al-Qutaish and Ain, 2010). Further development was taken up by the ISO, which issued the ISO/IEC 9126 standard in 1991. The ISO 9126 provides a hierarchical model consisting of six characteristics and 21 subcharacteristics. It draws from previously developed standards, and is applicable for every kind of software. ISO 9126 was followed by the current version of the standard, ISO/IEC 25010:2011, which expands to 8 characteristics and 31 subcharacteristics. As an example, *Maintainability* is one of the 8 characteristics, having subcharacteristics *Modularity*, *Reusability*, *Analysability*, *Modifiability* and *Testability*.

2.3 Related Work

The importance given to software metrics is illustrated by the number of works that aim to build a relation between metric values and software quality. (Chidamber et al., 1998) explore the relation between CK metrics and productivity, rework and design effort for managerial use. In (Li and Henry, 1993b), authors employ object-oriented and size metrics to investigate whether metrics can be used as a predictor of maintenance effort, which they define as the number of source code lines added, deleted or modified at class level. Their case study consisted of three years worth of maintenance changes for two Ada systems. They confirmed the existence of a relation between metric values and class-level changes in source code. In (Basili et al., 1996), authors employ the same metrics to assess them as predictors of fault-proneness using eight systems developed using C++. They adapt CK metric definitions to C++ and empirically validate that all CK metrics, with the exception of LCOM to be good fault predictors. In the case of LCOM, authors theorize that the definition provided by CK is not conductive for detecting faults induced by coupling. Another intriguing result is the inverse relation found between NOC and fault proneness. This is explained knowing that reused classes are less fault prone, as reported by existing research (Melo et al., 1995). (Tang et al., 1999) employ three C++ systems with a known fault history to validate the use of CK metrics, together with proposing additional object-oriented metrics geared towards identifying fault-prone classes. Among established metrics, RFC and WMC are indicated as suited to the task. CK metrics are again used by (Gyimothy et al., 2005) as fault predictors within the Mozilla web and mail open-source suite, using the project's Bugzilla repository as ground truth. Authors report that CBO and LOC reliably indicate fault-prone classes, while DIT and NOC are untrustworthy. In (Xu et al., 2008), authors indicate LOC, WMC, CBO and RFC to be reliable in defect estimation when applied on a public NASA data set, of which LOC appeared most reliable. However, they consider further research to be required to identify further relations between metric values, as well as those with dependent variables.

A number of works link object-oriented metrics with quality attributes as defined within ISO standards. (Kanellopoulos et al., 2010) links ISO 9126 software characteristics together with CK and other object-oriented metrics and undertakes an experimental evaluation using two open-source Java software servers. In (Molnar and Motogna, 2017), authors propose a model based on the values for CBO, DIT, WMC and ILCOM to describe changes to maintainability as defiend by ISO 9126 and validate the proposed model in a longitudinal study of open source software. (Mohd and Khan, 2012) studies the understandability characteristic, which is modeled as a linear combination of coupling, cohesion and inheritance, with coefficients and representative metrics assigned to each. Interestingly, it is one of the approaches where metric values are derived from class diagrams and not directly using the source code. (Dandashi, 2002) demonstrates a method to assess direct quality attributes using indirect ones. The paper also proposes relations between object-oriented metrics and quality attributes. (Elish and Alshayeb, 2012) continue the work presented by (Dandashi, 2002) and study the six external quality attributes as defined by the ISO standard and the correlation between them and several software metrics. The paper is focused on undertaking refactoring without affecting software quality, as well as refactoring with the purpose of improving specific quality attributes. In (Gyimothy et al., 2005), authors study the relation between object-oriented metrics and software reliability and evaluate it using open source software.

(Barkmann et al., 2009) carry out a large-scale cross-sectional study of 146 Java applications. The methodology is described in detail. Projects are downloaded from open-source software repositories, imported into an IDE and compiled to ensure the absence of syntax errors and missing dependencies. Information about 16 metrics linked with product qual-

ity is extracted using the VizzMaintenance tool. Authors present metric distributions, descriptive statistics and calculate metric dependencies.

While several presented papers study metric values and correlations, they are geared towards identifying a relation between metric values and software quality characteristics, such as those defined by the ISO standards. We believe that further work needs to be undertaken to ensure the validity of these conclusions. This is especially true given the findings in (Emam et al., 2001), where authors show that class size affects metric correlation values. Their experimental study involving a large scale framework shows that in many cases, strong correlations are the result of larger class sizes, instead of some other property.

Furthermore, a prevalent issue regards the use of different definitions for metrics, tools to extract them and target applications under study. This makes follow-up studies difficult to undertake, and comparison between studies impossible. As such, our objectives included ensuring that the methodology, tooling and results we obtained facilitate follow-up examination. The present paper discusses our methodology, tooling, as well as our most important findings. Our data set, including processed results for all considered metrics are available on our website[6]. Furthermore, we ensure that our study methodology and software tooling is compatible with that used in (Barkmann et al., 2009). As such, we compare and evaluate our results with those already presented. This enables us to employ results of existing research and draw stronger conclusions.

3 EVALUATION

3.1 Target Applications

Selection of target applications was guided by several criteria. First, we wanted our evaluation to be representative for a large number of existing systems. Second, given the popularity of large-scale, widely used open source projects, as well as our requirement of having free access to application source code, we settled on employing several such systems. We searched for applications that are easy to set up and which come without complex dependencies. This was important in the context of existing research (Barkmann et al., 2009) that reported that over one third of downloaded projects required additional work in order to properly compile and run. Last but not least, we searched projects with a long development history, and with a large number of available versions. This allows us to carry out a longitudinal study, an observational research method in which metric data is collected from each available version of the studied applications, over a significant period of time. This is rather difficult in the context of open-source software, as many projects with a development time frame of years undergo development hiatuses.

The applications chosen for our evaluation are the jEdit[7] text editor, the FreeMind [8] mind mapping application and the TuxGuitar[9] tablature editor. All three are GUI-driven applications developed in Java and hosted on SourceForge [10].

jEdit is an open-source text editor in development since 2000. The system already served as target application for previous research in application testing (Arlt et al., 2012; Yuan and Memon, 2010). jEdit has a large number of users, having over 80k downloads in 2018, and reaching over 8.9 million downloads in its 18 years of existence[11]. jEdit also has plugin support, but no plugin source code was included in our study.

FreeMind is a widely used mind-mapping application. The application had 775k downloads in 2018 and similar to jEdit, it was also employed for empirical evaluation in software research (Arlt et al., 2012). The source code evaluated in our study was downloaded from the project website and did not include plugin code.

TuxGuitar is a multitrack guitar tablature editor that provides features for importing and exporting various tablature formats. TuxGuitar is also a popular application, having over 229k downloads for 2018. In contrast to jEdit and FreeMind, TuxGuitar source code is bundled with code for importing and exporting from various data formats, implemented in the form of plugins. Since the code was included with the application distribution, and it provides core functionality for the system, it was included in our evaluation. Table 1 provides information regarding the size of the first and last version of the evaluated target systems, as an indication of their complexity.

All three projects have a full development log on SourceForge, allowing us to analyze all released versions. In this paper, we are interested in a long-term study of the applications. We analyze all their released versions, leading to 43 versions for jEdit, 38 for FreeMind and 26 for TuxGuitar.

[6] http://www.cs.ubbcluj.ro/ se/enase2019

[7] http://jedit.org

[8] http://freemind.sourceforge.net/wiki/index.php/Main_Page

[9] http://www.tuxguitar.com.ar

[10] https://sourceforge.net

[11] Download data points taken on December 23rd, 2018

Table 1: First and last studied version of each target application.

Application	Version	LOC	Classes
jEdit	2.3.pre2	33,768	322
	5.5.0	151,672	952
FreeMind	0.0.3	3,722	53
	1.0.Beta2	63,799	587
TuxGuitar	0.1pre	11,209	122
	1.5.2	108,495	1,618

All 107 resulting versions were imported into an IDE. Each version was manually checked for the inclusion of library source code; when found, such code was separated from application code into library files on the application classpath, to ensure that it does not distort measurement results. As applications were tested under the Java 8 platform, several compilation errors had to be addressed in the case of older application versions developed under earlier platform versions. This was done with consideration not to affect resulting metric values. Each application version was started, and its functionalities were thoroughly tested in order to ensure all required code was included. Raw metric data was exported into spreadsheet format and is available on our website. Scripts were developed to extract and aggregate metric and dependency information. This was done for every application version, as well as aggregated over each application as well as overall.

3.2 Descriptive Statistics

We considered five data sets within our evaluation. The first one is the (Barkmann et al., 2009) data set, consisting of a large cross-sectional study of singular versions from 146 applications. The aggregated data for each of our three studied applications resulted in three per-application data sets. Finally, the aggregated data from all 107 studied application versions resulted in the overall data set. For the purposes of brevity, the present section discusses the overall data set, and references per-application information only where required.

Figure 1 illustrates metric histograms and the overall data set, including descriptive statistics from (Barkmann et al., 2009). The first observation is that metric distribution is similar in both studies, with none of the metrics having a normal distribution. With regards to extreme values, minimums are always 0, the lowest possible value, except for the LOC metric, where it is 1. Differences between maximum values reported by us and (Barkmann et al., 2009) are as expected much larger and due to outliers. These observations also hold for the application specific data sets. Furthermore, metrics follow the distribution shown in Figure 1 in each version of the studied applications.

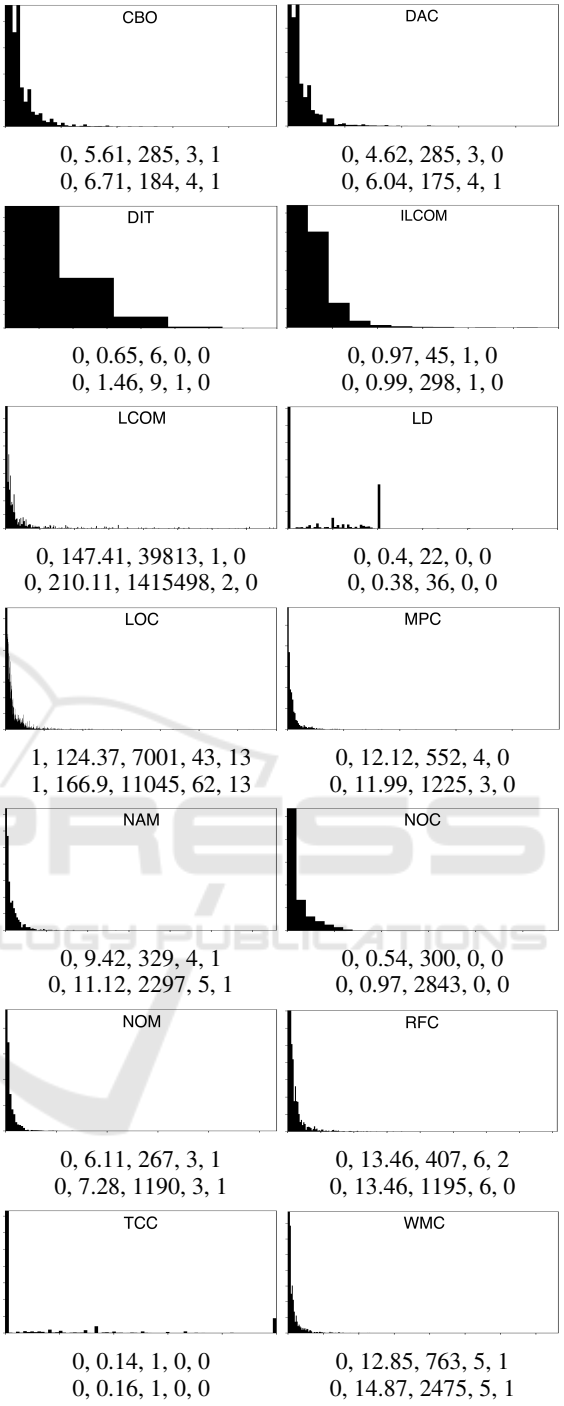

Figure 1: Each histogram: minimum, mean, maximum, median, modus. Our results on top row, results from (Barkmann et al., 2009) on bottom row for comparison.

Exploring the values for mean, median and modus

Table 2: Mean metric values per application.

Application	CBO	DAC	DIT	ILCOM	LCOM	LD	LOC	MPC	NAM	NOC	NOM	RFC	TCC	WMC
FreeMind	5.36	4.21	0.78	0.99	197.61	0.48	108.61	10.92	9.75	0.64	6.87	13.54	0.13	12.50
jEdit	4.66	4.09	0.41	0.77	124.83	0.34	156.43	9.46	8.40	0.37	5.15	10.62	0.15	13.40
TuxGuitar	7.32	6.07	0.87	1.25	130.81	0.40	90.96	17.48	10.67	0.70	6.80	17.78	0.15	12.36

values proves more interesting. First, median and modus values are very close in all five data sets, with those presented in Figure 1 representative of application specific data. Mean values however show more variability. Table 2 shows mean metric values for each application data set. Mean values for CBO, NAM, NOM, TCC and WMC are close across all data sets. However, the data for ILCOM, MPC and RFC can be confounding. While overall aggregate results are similar in our data set and (Barkmann et al., 2009), they also indicate that application specific differences exist.

3.3 Longitudinal Evaluation

In this section we discuss how metric values change during target application development. Due to space considerations, we eschew from including all descriptive statistical data points[12], which can be found on our website.

The data points presented in Figure 1 were computed for every metric and application version. Values follow the distributions already presented for all metrics and application versions. Minimum and median values are close to the overall ones presented in the table. Maximum values show much greater variability, but since these are represented by outliers we do not detail them.

The rest of this section is dedicated to a discussion of mean metric values. We believe our following observations are best accompanied by a description of the minimum and maximum mean metric values, as illustrated in Table 3. For each application, data under the *all* header includes all application versions, while data on the right-hand side was calculated taking into consideration only the given versions. In the case of FreeMind, these are versions later than, but including 1.0.0Alpha4.

For each application, we examined the changes to mean values across the versions, and corroborated the information with manual source code analysis. Each application version was also run in order to determine existing functionalities, as well as changes from previous versions.

The first observation is that most significant changes to metric values are linked with the development of the earliest application versions, before their full feature set is implemented. This is most readily observable for FreeMind and TuxGuitar, as their earliest versions (0.0.3 and 0.1pre respectively) are buggy and lack a number of features prominent in future versions. For jEdit, the first public version is 2.3pre2 and it appears more feature complete and undergoes less significant changes. This is reflected both at application functionality as well as metric value level.

The second observation is that for all three applications, there exist versions where mean metric values are greatly disrupted. Source code examination revealed two culprits. The first are changes to application functionality. For jEdit, version 2.5 adds support for FTP and virtual file systems, functionalities implemented using a small number of classes having high LOC and WMC. Many versions also include significant changes for the management of the text-editing area, such as versions 4.0pre4, 4.3pre8, and 4.3pre16. Version 5.0 also incorporates important changes, with updates to most of the source code packages. Many versions also included or updated complex code for custom GUI components, such as versions 4.3pre4 and 4.3pre16. This situation is similar in the case of FreeMind and TuxGuitar. The second culprit is represented by versions that include important code refactorings, which in many cases were not accompanied by functional changes. This is the case for jEdit 2.4.2, where 21 classes having high LOC and WMC values used to parse source code were replaced with a unified implementation based on a general parser and XML descriptors. Similarly, in version 3.0final, 153 event handler classes with low LOC and WMC were replaced with a centralized handler implementation. We find similar changes in FreeMind as well as TuxGuitar. We acknowledge that observed modifications raise the question of how to measure source code developed in several languages, not all of which use an imperative paradigm. However, while an important topic in itself, it is beyond the scope of our current work.

Our third observation is that once the application architecture is stable and its core set of functionalities is implemented, the variation in metric values is significantly reduced. This is also illustrated in Table 3. For FreeMind and TuxGuitar, we considered the earliest public release of versions 1.0 to be the first *mature* version. As the first public release of jEdit was version 2.3pre2, we studied the development and fea-

[12] 107 application versions x 14 studied metrics x 5 data points = 7490 data points.

Table 3: Extreme values for metric means per all application versions (left-hand side) and *mature* versions (right-hand side).

	FreeMind				jEdit				TuxGuitar			
	all		after 1.0.0Alpha4		all		after 4.0pre4		all		after 1.0rc1	
Metric	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
CBO	3.89	6.15	5.33	5.57	3.85	4.91	4.29	4.91	6.03	7.88	7.06	7.88
DAC	2.67	5.30	4.20	4.38	3.45	4.30	3.77	4.30	4.76	6.97	5.16	6.97
DIT	0.15	1.69	0.70	1.03	0.32	0.70	0.32	0.43	0.45	1.07	0.78	1.07
ILCOM	0.81	1.04	0.99	1.04	0.49	0.83	0.79	0.83	1.07	1.46	1.15	1.46
LCOM	84.85	237.90	196.85	237.90	43.44	149.31	126.79	149.31	90.94	176.79	117.15	176.79
LD	0.30	0.52	0.48	0.51	0.23	0.37	0.34	0.37	0.35	0.50	0.35	0.50
LOC	63.35	157.84	100.05	110.79	91.29	177.37	158.64	177.37	73.13	116.69	73.13	115.25
MPC	6.99	13.20	10.59	10.92	6.79	10.05	9.34	10.05	14.26	22.85	14.65	22.85
NAM	7.06	10.09	9.85	10.09	5.18	9.19	8.53	9.19	9.41	12.98	9.41	12.98
NOC	0.15	1.44	0.59	0.63	0.29	0.65	0.29	0.38	0.45	0.92	0.58	0.92
NOM	5.26	7.06	6.88	6.99	3.16	5.49	5.28	5.46	6.13	8.13	6.13	8.13
RFC	9.74	15.17	13.49	13.62	7.91	11.14	10.39	11.14	14.81	22.21	15.80	22.21
TCC	0.03	0.16	0.14	0.16	0.06	0.17	0.14	0.17	0.12	0.22	0.12	0.18
WMC	8.52	14.41	12.32	12.55	8.52	15.05	13.43	15.05	10.63	15.38	10.63	15.38

ture history and settled on version 4.0pre4, released January 2002 as its first mature release. Table 3 illustrates that in the case of mature application versions, the range of variation of mean metric values is narrowed. Furthermore, we observe that trends exist in the case of every applications. In our opinion, this is a strong indication that further case studies focused on longitudinal evaluation need to be undertaken, as cross-sectional studies tend to coalesce all values and lose application-specific context.

Our final observation is related to an expectation that mean metric values increase as applications become more complex. While mitigated by functional changes and refactorings, we find this to be true in the case of FreeMind and jEdit, especially for the size metrics LOC, NAM and NOM. For TuxGuitar, the observation holds until version 1.3.0, where a large number of small-complexity classes were added to the system, lowering mean metric values. For example, mean LOC decreases from 115 in version 1.2.0 to 80 in 1.3.0, while mean NAM decreases from 12.96 to 9.85. This illustrates that first of all, statistical data by itself is of limited value when not complemented by understanding the application context. Second of all, it shows that application architecture has important, measurable effects.

3.4 Dependent Metrics

Our goal is to identify dependent metrics and compare our results with (Barkmann et al., 2009)'s cross-sectional study. We aim to identify which metric pairs show consistent correlation across applications and application versions, as well as identify non-dependent metrics. Furthermore, as existing research identifies class size to have a confounding effect on metric dependency (Emam et al., 2001), we account for class sizes by partitioning application classes into quartiles.

As illustrated in Figure 1, none of the studied metrics are normally distributed. As such, the Spearman rank correlation was used to calculate correlation information. Table 4 presents per application correlation data, with strong correlations highlighted. We also included data from (Barkmann et al., 2009), as it is directly comparable.

We observe that our results are in agreement with those of (Barkmann et al., 2009). Strong correlation exists between certain metrics expressing coupling (CBO and DAC), size (LOC, NOM and NAM) and complexity (MPC, RFC and WMC). Inheritance metrics DIT and NOC are not dependent with any other metrics, including each other. LD and TCC show much stronger correlation in (Barkmann et al., 2009) than in our study. Meanwhile, we find LCOM to be correlated with both size (LOC, NAM and NOM) as well as complexity (MPC, RFC and WMC) metrics.

The LOC metric correlates with both size as well as complexity metrics. Given the findings of (Emam et al., 2001), the next step was to ascertain the effect of class size on identified correlations. We partitioned class sizes to quartiles and calculated the dependency below the first quartile (below Q1), in the inter-quartile range as well as above the third quartile (above Q3). The LOC metric was omitted, as it was already used to partition the data. Table 5 presents the mean metric correlations obtained. As the domain for Spearman's rank correlation is $[-1,1]$, mean correlations were calculated by summing the absolute values of correlation. Our observation confirms the results

Table 4: Correlation between studied metrics in FreeMind (top row), jEdit (second row), TuxGuitar (third row) and as reported in (Barkmann et al., 2009) (bottom row).

Metric	CBO	DAC	DIT	ILCOM	LCOM	LD	LOC	MPC	NAM	NOC	NOM	RFC	TCC	WMC
DAC	**0.978**													
	0.988	1.000												
	0.969													
	0.982													
DIT	0.289	0.303												
	0.182	0.202	1.000											
	0.186	0.100												
	0.529	0.528												
ILCOM	0.460	0.496	0.086											
	0.446	0.464	-0.003	1.000										
	0.075	0.111	-0.296											
	0.539	0.414	0.391											
LCOM	0.530	0.562	0.054	0.559										
	0.559	0.562	-0.038	0.406	1.000									
	0.209	0.210	-0.126	0.376										
	0.539	0.551	0.405	0.478										
LD	0.200	0.221	0.076	0.403	0.111									
	0.184	0.211	0.153	0.561	0.070	1.000								
	0.031	0.068	-0.206	0.437	0.114									
	0.315	0.334	0.430	0.794	0.449									
LOC	0.587	0.617	0.090	0.562	0.770	0.257								
	0.773	0.781	-0.001	0.554	**0.848**	0.215	1.000							
	0.467	0.460	-0.146	0.345	0.667	0.160								
	0.581	0.600	0.142	0.477	0.580	0.325								
MPC	**0.837**	**0.817**	0.229	0.468	0.604	0.173	0.661							
	0.837	**0.828**	0.064	0.448	0.759	0.150	**0.871**	1.000						
	0.620	0.567	0.035	0.185	0.564	0.042	**0.825**							
	0.830	**0.813**	0.534	0.576	0.597	0.505	0.667							
NAM	0.690	0.729	0.111	0.720	**0.867**	0.329	**0.850**	0.714						
	0.711	0.720	-0.010	0.653	**0.850**	0.292	**0.947**	**0.829**	1.000					
	0.301	0.305	-0.230	0.571	0.783	0.290	0.788	0.597						
	0.519	0.533	0.165	0.632	0.682	0.468	**0.837**	0.627						
NOC	-0.005	0.021	-0.035	0.106	0.148	0.014	0.062	0.021	0.133					
	-0.046	-0.031	-0.050	0.028	0.014	-0.005	0.017	-0.028	0.014	1.000				
	-0.029	-0.036	-0.062	0.024	0.014	0.023	-0.028	-0.026	0.015					
	0.061	0.087	0.406	0.572	0.380	0.620	-0.110	0.216	0.061					
NOM	0.564	0.600	0.104	0.658	**0.913**	0.231	**0.823**	0.636	**0.954**	0.163				
	0.688	0.693	-0.055	0.590	**0.907**	0.207	**0.947**	**0.841**	**0.966**	0.032	1.000			
	0.327	0.334	-0.231	0.559	**0.830**	0.274	**0.836**	0.675	**0.928**	0.032				
	0.563	0.580	0.238	0.598	0.799	0.480	0.791	0.650	**0.911**	0.144				
RFC	0.746	0.748	0.185	0.625	**0.842**	0.234	**0.804**	**0.881**	**0.911**	0.114	**0.907**			
	0.830	**0.824**	0.023	0.535	**0.823**	0.189	**0.929**	**0.961**	**0.914**	-0.004	**0.935**	1.000		
	0.539	0.494	-0.028	0.324	0.623	0.123	**0.880**	**0.924**	0.733	-0.003	**0.821**			
	0.717	0.709	0.277	0.529	0.715	0.019	**0.801**	**0.817**	**0.837**	0.022	**0.907**			
TCC	0.020	0.025	0.021	0.112	-0.042	0.226	0.039	0.040	0.051	-0.022	0.021	0.043		
	0.058	0.070	0.051	0.252	-0.003	0.430	0.095	0.066	0.126	-0.049	0.087	0.081	1.000	
	0.081	0.094	-0.054	0.040	-0.053	0.258	0.039	-0.001	0.078	-0.050	0.026	0.011		
	0.336	0.355	0.543	0.781	0.468	**0.803**	0.269	0.510	0.411	**0.841**	0.455	0.367		
WMC	0.534	0.559	0.087	0.618	**0.861**	0.232	**0.891**	0.693	**0.903**	0.124	**0.932**	**0.904**	0.047	
	0.708	0.708	-0.049	0.530	**0.888**	0.160	**0.959**	**0.879**	**0.933**	0.003	**0.964**	**0.939**	0.081	1.000
	0.388	0.386	-0.173	0.373	0.725	0.160	**0.950**	**0.820**	0.792	-0.009	**0.883**	**0.881**	0.009	
	0.599	0.606	0.204	0.570	0.725	0.440	**0.844**	0.712	**0.880**	0.054	**0.939**	**0.930**	0.405	

in (Emam et al., 2001), as class size has an important effect on correlation strength. This is especially observable in the case of size and complexity metrics, and less so for coupling and inheritance metrics.

At this point, we have identified metric pairs having strong dependency, and checked our results with those obtained by (Barkmann et al., 2009). Given the longitudinal nature of our data, we were also interested in checking metric correlation strength across application versions. In broad terms, our observations mirror those presented in the previous section. Metric correlations show greater variability in earlier application versions, and tend to stabilize once applications mature. Strongly dependant metric pairs exhibit this behaviour across all applications and application versions. In the case of metric pairs having mean correlation over 0.8 for each application, the minimum correlation in a single application version was above

Table 5: Mean metric correlations partitioned by class size (LOC). Top row represents classes below Q1, middle row is inter-quartile range, bottom row represents classes above Q3.

Metric	FreeMind	jEdit	TuxGuitar
CBO	0.25	0.29	0.34
	0.30	0.29	0.35
	0.44	0.50	0.23
DAC	0.16	0.21	0.29
	0.36	0.25	0.30
	0.45	0.50	0.21
DIT	0.20	0.15	0.32
	0.18	0.13	0.31
	0.10	0.09	0.11
ILCOM	0.23	0.16	0.22
	0.20	0.23	0.35
	0.38	0.35	0.28
LCOM	0.33	0.24	0.29
	0.28	0.18	0.33
	0.51	0.52	0.39
LD	0.21	0.15	0.14
	0.24	0.20	0.21
	0.07	0.06	0.13
MPC	0.25	0.26	0.35
	0.26	0.26	0.31
	0.46	0.54	0.40
NAM	0.39	0.20	0.28
	0.41	0.25	0.32
	0.56	0.57	0.41
NOC	0.17	0.10	0.11
	0.11	0.07	0.10
	0.12	0.03	0.10
NOM	0.36	0.27	0.31
	0.39	0.30	0.40
	0.53	0.57	0.46
RFC	0.35	0.30	0.36
	0.44	0.34	0.29
	0.56	0.58	0.42
TCC	0.15	0.07	0.16
	0.14	0.16	0.11
	0.12	0.09	0.10
WMC	0.33	0.26	0.30
	0.38	0.27	0.33
	0.49	0.56	0.41

0.7. When accounting for the confounding effect of class size measured as LOC, correlations remained strong for all identified metric pairs. This held true both for classes having LOC below Q1 as well as in the inter-quartile range. We also studied metric pairs that appear to be uncorrelated. Our observations are similar, but in reverse, as these metric pairs do not exhibit strong correlation within any of the studied application versions. In this case, class size did not seem to affect measured correlations.

4 THREATS TO VALIDITY

Our study was carried out in four steps: application preparation, data extraction, data processing and analysis. We documented all steps required to replicate our study. All intermediate and complete final results are available on our website. We ensured that influencing factors did not exist beside the analyzed source code. Exact definitions were provided for the employed metrics. Each application version was compiled and executed, and its functionalities manually checked, to ensure that source code was complete. Given that different extraction tools can yield different results (Bakar and Boughton, 2012; Awang Abu Bakar and Boughton, 2008), we double-checked extracted metric values using the MetricsReloaded tool.

To limit external threats to validity, we selected three GUI-based applications developed in the same programming language. Applications were prepared and data extraction finalized before processing, to ensure the absence of selection bias. We presented the most important results both separated by application, as well as in aggregate form. We compared them with a relevant large-scale, cross-sectional study of open-source software that used a well defined methodology. However, we believe that more research is required before our conclusions can be generalized to other application types, such as libraries or non-GUI applications. Furthermore, our study remains limited by the selection of programming language, metric definition and application types.

5 CONCLUSIONS AND FUTURE WORK

We carried out an exploratory evaluation regarding the values of widely used software metrics, as well as the relation between these values in the context of three complex open-source, GUI-driven applications. We analyzed our results for the entire development history of the studied applications, and evaluated all publicly released application versions. We structured our study to ensure it is repeatable and evaluated our results in the context of a comparable large-scale evaluation. The combined results aggregate metric data from over 250 application versions[13] and provide a sound foundation for further research.

The first conclusion is that metric value distributions were consistent across the studies. Furthermore, there is some similarity regarding median and mean values. This is true especially when examining the mature application versions in our study. We believe our work can be used as a starting point to determine metric threshold values indicating good design prac-

[13](Barkmann et al., 2009) evaluated 146 software projects

tices. Given the variation in metric values between early and current application versions, we believe longitudinal studies to provide valuable contributions in this regard.

The second conclusion is that strongly dependent metric pairs can be identified. They are the same both in our longitudinal evaluation as well as the referred cross-sectional one. Our longitudinal examination has shown these relations to be extremely stable across all application versions, including the earliest ones. These relations proved to be impervious to the effects of class size. Their existence should be considered when building software quality models based on metric values. They can be used to select those metrics that best express a system property, or to avoid introducing undesired collinearity.

Our third conclusion regards the differences between the trends in metric values and dependencies between studied applications. Given that cross-sectional studies are unable to capture this, it strengthens the importance of longitudinal studies.

We aim to extend our research to other application types, including mobile as well as applications where user interface code is not dominant. Our goal is to study whether metric thresholds indicative of good design and development practices can be established. Furthermore, we aim to extend our research to applications developed using different platforms, and study the effect of the programming language on metric values. The main goal is to establish a metric-based model for software quality. While such attempts have already been undertaken, they are not based on a solid foundation of understanding the software development process and its outcomes, narrowing their range of application.

REFERENCES

Al-Qutaish, R. E. and Ain, A. (2010). Quality Models in Software Engineering Literature: An Analytical and Comparative Study. Technical Report 3.

ARISA Compendium - Understandability for Reuse (2018). http://www.arisa.se/compendium/node39.html#property:UnderstandabilityR. accessed November, 2018.

Arlt, S., Banerjee, I., Bertolini, C., Memon, A. M., and Schaf, M. (2012). Grey-box gui testing: Efficient generation of event sequences. *CoRR*, abs/1205.4928.

Awang Abu Bakar, N. and Boughton, C. V. (2008). Using a combination of measurement tools to extract metrics from open source projects. In Khoshgoftaar, T., editor, *Proceeding (632) Software Engineering and Applications - 2008*, pages 130–135. ACTA Press, Canada.

Bakar, N. S. A. A. and Boughton, C. V. (2012). Validation of measurement tools to extract metrics from open source projects. In *2012 IEEE Conference on Open Systems*, pages 1–6.

Barkmann, H., Lincke, R., and Löwe, W. (2009). Quantitative evaluation of software quality metrics in open-source projects. In *2009 International Conference on Advanced Information Networking and Applications Workshops*, pages 1067–1072.

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

Bassam Al-Badareen, A., Selamat, H., Jabar, M. A., Din, J., and Turaev, S. (2011). *Software Quality Models: A Comparative Study*, volume 179. Springer, Berlin, Heidelberg.

Chidamber, S. R., Darcy, D. P., and Kemerer, C. F. (1998). Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639.

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

Dandashi, F. (2002). A method for assessing the reusability of object-oriented code using a validated set of automated measurements. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 997–1003, New York, NY, USA. ACM.

DeMarco, T. (1986). *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Elish, K. O. and Alshayeb, M. (2012). Using software quality attributes to classify refactoring to patterns. *Journal of Software*, pages 408–419.

Emam, K. E., Benlarbi, S., Goel, N., and Rai, S. N. (2001). The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650.

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

Hitz, M. and Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27.

Kanellopoulos, Y., Antonellis, P., Antoniou, D., Makris, C., Theodoridis, E., Tjortjis, C., and Tsirakis, N. (2010). Code quality evaluation methodology using the ISO/IEC 9126 standard. *International Journal of Software Engineering & Applications*, 1(3):17–36.

Landman, D., Serebrenik, A., and Vinju, J. (2014). Empirical analysis of the relationship between cc and sloc in a large corpus of java methods. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 221–230, Washington, DC, USA. IEEE Computer Society.

Lenhard, J., Blom, M., and Herold, S. (2018). Exploring the suitability of source code metrics for indicating architectural inconsistencies.

Li, W. and Henry, S. (1993a). Maintenance metrics for the object oriented paradigm. In *[1993] Proceedings First International Software Metrics Symposium*, pages 52–60.

Li, W. and Henry, S. (1993b). Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122.

Lincke, R., Lundberg, J., and Löwe, W. (2008). Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*.

Marinescu, R. (2005). Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704.

Melo, W. L., Briand, L. C., and Basili, V. R. (1995). Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems. Technical report.

Mohd, N. and Khan, P. R. (2012). An empirical validation of understandability quantification model. *Procedia Technology*, 4.

Molnar, A. and Motogna, S. (2017). Discovering maintainability changes in large software systems. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, IWSM Mensura '17, pages 88–93, New York, NY, USA. ACM.

Ott, L., Bieman, J. M., Kang, B.-K., and Mehra, B. (1970). Developing Measures of Class Cohesion for Object-Oriented Software. Technical report.

Rodriguez, D. and Harrison, R. (2001). An overview of object-oriented design metrics.

Sarker, M. (2005). An overview of object oriented design metrics. *Umeå University, Sweden*.

Silva, R. and Costa, H. (2015). Graphical and statistical analysis of the software evolution using coupling and cohesion metrics - An exploratory study. In *Proceedings - 2015 41st Latin American Computing Conference, CLEI 2015*.

Succi, G., Pedrycz, W., Djokic, S., Zuliani, P., and Russo, B. (2005). An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite. *Empirical Software Engineering*.

Tang, M.-H., Kao, M.-H., and Chen, M.-H. (1999). An empirical study on object-oriented metrics. In *Proceedings of the 6th International Symposium on Software Metrics*, METRICS '99, pages 242–, Washington, DC, USA. IEEE Computer Society.

Xu, J., Ho, D., and Capretz, L. F. (2008). An empirical validation of object-oriented design metrics for fault prediction. *Journal of Computer Science*.

Yuan, X. and Memon, A. M. (2010). Generating event sequence-based test cases using GUI run-time state feedback. *IEEE Transactions on Software Engineering*, 36(1).