Evaluation of Software Product Quality Metrics

Arthur-Jozsef Molnar(✉), Alexandra Neamţu, and Simona Motogna

Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania
{arthur,motogna}@cs.ubbcluj.ro
nais1841@scs.ubbcluj.ro

Abstract. Computing devices and associated software govern everyday life, and form the backbone of safety critical systems in banking, healthcare, automotive and other fields. Increasing system complexity, quickly evolving technologies and paradigm shifts have kept software quality research at the forefront. Standards such as ISO's 25010 express it in terms of sub-characteristics such as maintainability, reliability and security. A significant body of literature attempts to link these subcharacteristics with software metric values, with the end goal of creating a metric-based model of software product quality. However, research also identifies the most important existing barriers. Among them we mention the diversity of software application types, development platforms and languages. Additionally, unified definitions to make software metrics truly language-agnostic do not exist, and would be difficult to implement given programming language levels of variety. This is compounded by the fact that many existing studies do not detail their methodology and tooling, which precludes researchers from creating surveys to enable data analysis on a larger scale. In our paper, we propose a comprehensive study of metric values in the context of three complex, open-source applications. We align our methodology and tooling with that of existing research, and present it in detail in order to facilitate comparative evaluation. We study metric values during the entire 18-year development history of our target applications, in order to capture the longitudinal view that we found lacking in existing literature. We identify metric dependencies and check their consistency across applications and their versions. At each step, we carry out comparative evaluation with existing research and present our results.

Keywords: Software metric · Software quality · Descriptive statistics · Cross-sectional study · Longitudinal study

1 Introduction

Software development has experienced an exponential increase over the past decades, which can be observed in the variety of applications available (such as

© Springer Nature Switzerland AG 2020
E. Damiani et al. (Eds.): ENASE 2019, CCIS 1172, pp. 163–187, 2020.
https://doi.org/10.1007/978-3-030-40223-5_8

web, mobile, real time and so on), as well as in application size and complexity. Large-scale and enterprise applications are being developed over longer periods of time, using larger teams that are in many cases geographically distributed. In the same time frame, project management and software methodologies, available tools and development environments have evolved in an attempt to keep the pace with increasing requirements.

This increase in size and complexity raises another problem, namely the necessity to control the software development processes, and implicitly to measure it, as "you cannot control that which you cannot measure" [11]. In accordance, the domain of software metrics has evolved both as methodology as well as in terms of available software products, being influenced by the development of programming languages, paradigms and methodologies.

Software quality assurance is also an important aspect as software products have to satisfy user needs related to ease of use, security and reliability. Furthermore, development related needs such as maintainability, portability and testability must also be accounted for. The latest software quality models have undergone standardization processes, such as ISO standards 9126 and 25010, in order to establish a set of common criteria for software products. These standards can significantly benefit from data provided by software metrics, as there exists consistent research results that report the influence of software metrics on software quality factors [8,17,22,25,35].

However, additional data analysis is required before general models can be built [5]. Also, even if the influence of metrics on quality factors is well understood and accepted, there does not yet exist any general accepted method to evaluate software quality factors based on software metric values. As such, the relation between metric values and software quality factors remains an open problem. We aim to address this issue in the present paper. We carry out a comprehensive evaluation on values of software metrics that are widely associated with software product quality. We employ methodology and tooling compatible with existing results in order to enable comparative evaluation. We carry out a long-term study targeting three complex, open-source applications, and provide the following contributions:

(i) A clear description of our methodology, metric definitions and tooling used to extract metric values. Doing this ensures that our results can be used for comparative evaluation in future studies. We made all extracted metric values publicly available[1].
(ii) A quantitative evaluation of metric values is carried out and detailed for all target application versions.
(iii) A longitudinal exploratory study that examines the evolution of metric values over the course of 18 years of target application development.
(iv) Identification of statistical correlations between metric pairs. We identify both strongly correlated metrics as well as metrics that appear independent. We account for the confounding effect of class size and examine the stability of the correlation strength across application versions.

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

(v) A comparative evaluation of metric values and statistical correlations between target applications. We identify trends in metric values and correlations that are application-specific, together with those that hold across the target applications.
(vi) An evaluation of our obtained results in the context of existing research that uses the same methodology and software tools.

One of our study's key contributions lies in the selection of target applications. Existing studies are built around one of the following two approaches. The first one is where a number of applications are selected, and for each of them several versions are studied [17,32]. The second one considers a large number of target applications, that in many cases are automatically downloaded from open-source repositories [5], with a cross-sectional study including all of them [5,18,19]. Our approach aims to complement existing research. We select a number of three open-source applications developed on the same platform, having comparable complexity and scope, and include all their released versions in our study. This results in a large number of application versions that ensures statistical significance. More so, our approach includes both initial application versions, which are sometimes very simple functionality-wise and bug-prone. We also include the latest application versions, that appear polished, have extensive features sets and a consistent user base. This enables us to study how metric values evolve together with the target applications, as well as to identify any existing trends that might be influenced by application development status.

Another important contribution regards careful selection of software metrics and extraction tools. As detailed in our initial evaluation [26], we selected the evaluated metrics in order to cover complexity, inheritance, coupling and cohesion [2,22] as important characteristics of object-oriented software. In addition, the studied metrics can be found in existing literature studying software product quality [18,19,26,32]. Selection of the right tools for metric value extraction is also important, as most metrics have more than one definition [4,21]. As such, comparative evaluation can be carried out only with existing research that employs the same metrics, and that uses the same tooling to extract metric values.

In our initial evaluation [26], we employed the VizzAnalyzer tool[2], as it provides formal definitions of the extracted metrics. In addition, using VizzAnalyzer allows us to compare our results with those reported in [5], where authors use the same tool to carry out a cross-sectional study of 146 open-source applications. In our extensive literature survey, we identified [5] as the only paper that clearly detailed the study methodology and tooling in order to allow a comparative evaluation to be carried out. Since our present paper employs the same methodology and tooling as our initial evaluation [26], the obtained results are directly comparable. In addition, in the present paper we explore the effect class size has on metric correlations across our target applications. We show that metric variability is greatest in early versions, before application architecture is well

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

established. Furthermore, we find that most significant changes to metric values occur across a small number of application versions, which we examine in detail.

2 Software Metrics

Evolution in the domain of software metrics was influenced by changes in the development of software, with increasingly specific metrics being proposed for the measurement of both software products as well as software processes. This is reflected in the appearance of software metric tools, both general and language dependent, stand-alone as well as integrated into IDEs in the form of plugins.

The oldest software metrics that remain widely used today include lines of code (LOC), number of functions or modules, and the number of comment lines. This was followed by proposed metrics to measure code complexity, such as cyclomatic complexity [23] and Halstead volume [14]. In turn, these were used to compute additional, more complex metrics such as the Maintainability Index [25]. The object oriented paradigm introduced new entities and relations, and these were reflected by several newly proposed metrics. The reference set of object-oriented metrics was defined by Chidamber & Kemerer (CK) [8], were implemented in most software metrics tools, and used in many subsequent studies. The lack of cohesion in methods (LCOM) metric deserves special mention, as it was refined from its original definition in [8] by Li and Henry [20], and then by Hitz and Montazeri [16]. While these changes were driven by a desire to better capture the essence of cohesion, LCOM values can only be compared when extracted using the same definition. Several tools are available to compute the CK metrics (and many more). Some of them are available as IDE plugins, such as Metrics2[3] for Eclipse, MetricsReloaded[4] for IntelliJ, NDepend[5] for .NET, or as standalone tools such as JHawk[6] or Sourcemeter[7]. Each of them employs its own implementation for metric computation, leading to different results for the same metric when extracted with different tools.

The metrics selected for our study were all computed using the VizzAnalyzer tool, that uses the definitions provided in [37]. Other studies [5,21] are based on the same tool, giving us the possibility to compare the obtained results. According to [22], object-oriented metrics measure one of the four internal characteristics essential to object orientation, namely coupling, inheritance, cohesion and structural complexity. We present the metrics used in our study, categorized according to the internal characteristics they aim to measure. We start with metrics dedicated to measuring **coupling**:

- *Coupling Between Objects (**CBO**,* $v_{CBO} \in [0, \infty) \cap \mathbb{Z}$) [28] - for class **c** is computed as the number of other classes that are coupled to it. Two classes

[3] http://metrics.sourceforge.net.
[4] https://plugins.jetbrains.com/plugin/93-metricsreloaded.
[5] https://www.ndepend.com/.
[6] http://www.virtualmachinery.com/jhawkprod.htm.
[7] https://www.sourcemeter.com/.

are coupled when methods declared in one class use methods or instance variables defined by the other class. CBO indicates the required effort to test and maintain a class.

- *Data Abstraction Coupling (**DAC**, $v_{DAC} \in [0, \infty) \cap \mathbb{Z}$)* [20] - measures when a class is used in the implementation of methods of another class or when it is the domain of its instance variables. VizzAnalyzer does not include platform classes in this measurement.
- *Message Pass Coupling (**MPC**, $v_{MPC} \in [0, \infty) \cap \mathbb{Z}$)* [28] - counts the number of methods from other classes that are called. It indicates the degree of dependency on the system's other classes.

The following metrics measure the **inheritance** characteristic:

- *Depth of Inheritance Tree (**DIT**, $v_{DIT} \in [0, \infty) \cap \mathbb{Z}$)* [28] - represents the length of the longest path from a given class to the root of the inheritance tree. DIT also accounts for multiple paths possible in the context of multiple-inheritance languages such as C++.
- *Number of Children (**NOC**, $v_{NOC} \in [0, \infty) \cap \mathbb{Z}$)* [28,31] - counts the immediate subclasses found in the inheritance tree for a given class.

System **cohesion** is measured using the following metrics:

- *Lack of Cohesion in Methods (**LCOM**, $v_{LCOM} \in [0, \infty) \cap \mathbb{Z}$)* [28] - represents the difference between the number of methods pairs that don't have, respectively have, instance variables in common. This uses the original definition of the metric [28].
- *Improvement to Lack of Cohesion in Methods (**ILCOM**, $v_{ILCOM} \in [1, \infty) \cap \mathbb{Z}$)* [16] - this employs the improved definition provided by Hitz and Montazeri. In several papers and software tools this is referred to as LCOM5.
- *Tight Class Cohesion (**TCC**, $v_{TCC} \in [0, 1] \cap \mathbb{Q}$)* [27] - defined as the ratio between the number of directly connected public methods in a class divided by the number of all possible connections between the public methods of that class.

We employ the following metrics that measure the **structural complexity** of classes:

- *Locality of Data (**LD**, $v_{LD} \in [0, 1] \cap \mathbb{Q}$)* [16] - represents the ratio between the data that is local to a class and all the data used by the class. VizzAnalyzer includes non-public and inherited attributes.
- *Number of Attributes and Methods (**NAM**, $v_{NAM} \in [0, \infty) \cap \mathbb{Z}$)* [28] - represents the total number of attributes and methods that are locally defined by the class. This includes static methods, but excludes constructors and inherited fields or methods.
- *Number of Methods (**NOM**, $v_{NOM} \in [0, \infty) \cap \mathbb{Z}$)* [28] - represents the number of methods locally defined in the class. $NAM - NOM$ gives the number of locally defined attributes.

– *Response For a Class (**RFC**, $v_{RFC} \in [0, \infty) \cap \mathbb{Z}$)* [28] - counts the number of methods that could be invoked as a response to a given message. RFC is the number of methods called by a given class.
– *Weighted Method Count (**WMC**, $v_{WMC} \in [0, \infty) \cap \mathbb{Z}$)* [28] - defined as the sum of the complexities of all methods of a given class. The complexity of a method is its McCabe cyclomatic complexity [23].

Finally, we also examine metrics related with code **documentation**:

– *Length of Class Name (**LEN**, $v_{LEN} \in [1, \infty) \cap \mathbb{Z}$)* - the length of the class name counted in characters.
– *Lack of Documentation (**LOD**, $v_{LOD} \in [0, 1] \cap \mathbb{Q}$)* - the ratio of missing comments in a given class. Each class should have one comment per class, and an additional one for each defined method. This metric ignores the structure and the content of the comments.

Beside these metrics, we also measured the Lines of Code (LOC), since it is considered a universal software metric that can be used across most programming languages and which gives basic information about the size of a project. The relation between object-oriented metrics and LOC is worthy of further investigation, especially as existing research showed that class size has a strong confounding influence on quality models based on metrics [12].

3 State of the Art

The increasing attention given to software metrics is proven by the large number of studies in this domain. In most cases, existing research is geared towards one of the following three main directions: definition and analysis of proposed software metrics, software metric application in refactoring, and studying the relation between software metrics and software quality models.

3.1 Metrics

New metrics are being defined in order to fine-tune the characteristics of software systems, and in order to better reflect the properties of source code and associated artefacts. Examples include approaches to improve estimation of the maintenance effort [30], in order to supersede existing measures such as the Maintainability Index [25] which was shown to be outdated [10,15,29]. Other studies propose new metrics to better capture system coupling or cohesion [1,9].

Special interest has been also given to studying inter-metric dependency and correlation. A large scale study [5] was carried out using 146 Java applications, with 16 metrics extracted using the VizzAnalyzer tool. Barkman et al. applied different descriptive statistic techniques in order to detect metric dependencies. Landman et al. [18] show that typical *getters* and *setters* can distort metric dependencies by artificially increasing dependency values. In [12], authors show that class size has a significant impact on metric correlation, using experimental

data from a large scale telecommunication framework. These results illustrate that in order to validate strong conclusions derived from data analysis based on metric values, further research needs to be carried out. This is expected to be of special importance in the case of large-scale projects that were developed over a long period of time.

3.2 Refactoring

One of the first applications of software metrics was to use the recorded values in order to detect design flaws that could be solved through refactoring.

The impact of four refactoring methods on several metrics is described in [7], based on the source code's abstract syntax tree representation. Another significant study [34] refers to the impact 10 refactoring methods have on different metrics, including the Maintainability Index, cyclomatic complexity, DIT, class coupling and LOC. Changes to maintainability and modifiability after refactoring are presented in [34] through an empirical evaluation. The experimental evaluations included in the aforementioned studies illustrate that, in the case of complex systems, refactoring plays an important role for easing maintenance and keeping system complexity under control. The decision of where and how to refactor can be taken based on extracted values of suitable software metrics.

3.3 Software Quality Models

In recent years, several contributions attempted to connect software metrics with software quality factors. A software quality model is a hierarchical set of software quality factors or characteristics, that are further decomposed in subfactors or subcharacteristics. The first software quality model was introduced in 1976 by McCall, to which Boehm and Dromey later proposed important contributions. These initial contributions were later standardized by the ISO in the form of two families of standards: first, the ISO 9126, which expressed software quality as a function of six characteristics, that were comprised of 31 subcharacteristics. The 9126 standard was updated in the form of ISO 25010[8], which expands to the 8 characteristics shown in Fig. 1.

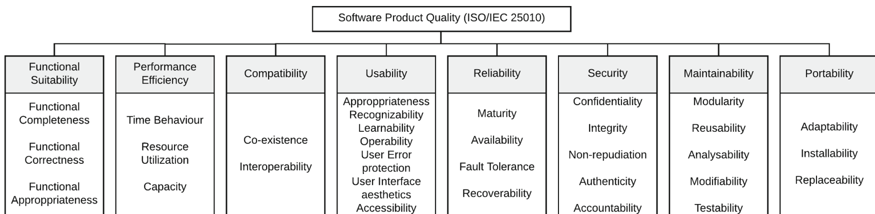

Fig. 1. ISO/IEC 25010 subcharacteristics hierarchy.

---

[8] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

Some of the factors, like Maintainability, are known to be highly influenced by coupling and cohesion, such as evaluated by the CBO, TCC and LCOM metrics. However, in many other cases, dependencies remain to be proven.

The ARISA Compendium [2] offers an exhaustive study of the influence of over 20 metrics on the software quality characteristics of ISO 9126. The authors' approach is based on linking metrics with those source code entities that are involved in the metric's formal description. In [20], authors claim that metrics should be adapted for each programming paradigm. They introduce object oriented metrics for the maintenance effort and validate their approach on two commercial systems using 10 metrics. A complementary study was carried out in [6], where the CK metrics are assessed in regard to fault proneness, with experiments performed on eight C++ applications. The study concluded that LCOM, as defined in the CK suite is not evidential for fault detection, but that the other CK metrics are well suited for predicting faults. Also, the experimental data revealed an inverse relation between NOC and faults, a result confirmed also by the impact of reuse on fault proneness presented in [24].

Another study [33] regarding the relation between CK metrics and faults evaluated the efficient selection of testing techniques. Authors reported RFC and WMC as the most suited metrics for this task. A similar study was conducted in [13] for the open-source Mozilla web and e-mail suite. It concluded that CBO and LOC are good predictors for faults, while DIT and NOC can lead to false results. An analysis [35] of CK metrics on a NASA public data set revealed that LOC, WMC, CBO and RFC can be safely used for defect estimation. The conclusion of the study recommended further investigation on the relation between metric values and different dependent variables using statistical and AI techniques.

4 Evaluation

4.1 Target Applications

In order to carry out our evaluation, the first step was to select target applications. We first established several required criteria. First, we decided to target open-source applications developed in Java that were user interface driven and which did not have significant dependencies on external libraries or databases. We also searched for applications having long-term, consistent development history that were freely available. Our goals required a longitudinal study, an observational research method that consists in setting up and collecting metric data from each of the application versions. As detailed in [5], this can prove difficult in the case of open-source software, where development effort suffers interruptions, and where there are no guarantees that all software versions are complete and usable. As such, we selected three popular applications with long development histories, which had an established user base as well as public development repositories populated since project inception. We also ensured selected applications were free from complex dependencies. This allowed us to run them in order to check that functionalities worked as expected in all application versions.

The selected applications are the FreeMind[9] mind mapper, the jEdit[10] text editor and the TuxGuitar[11] tablature editor. The entire development history of these applications can be found on SourceForge[12].

FreeMind. Is a mind-mapping application that found many uses in productivity and content management. FreeMind was also employed in previous software research [3]. It is also a popular application with a solid user base, having over 465k[13] downloads in 2019. FreeMind includes a plugin ecosystem with many plugins available. However, only the source code of the base application was included in our study.

jEdit. Is an open-source text editor, developed entirely using the Java programming language. It is also a popular system under test for other research endeavours in software testing [3,36]. jEdit is one of the popular SourceForge applications, having over 59k downloads in 2019 and reaching over 8.9 millions downloads in its 19 years of existence. Similar to the case of FreeMind, plugin code was not included in our evaluation.

TuxGuitar. Is a free, open-source multitrack guitar tablature editor with an SWT-based user interface. It includes features like multiple format data import and export, tablature and score editing. TuxGuitar is also a popular application having over 131k downloads in 2019. In contrast with FreeMind and jEdit, where we disregarded the applications' plugin ecosystems, in the case of TuxGuitar functionalities related to data import and export itself were implemented in the form of a plugin, and were included in our evaluation.

Table 1 provides information about the earliest and latest application versions included in our evaluation, indicating their change of complexity during the considered period.

Table 1. First and last studied version of each target application (from [26]).

Application	Version	LOC	Classes
jEdit	2.3pre2	33,768	322
	5.5.0	151,672	952
FreeMind	0.0.3	3,722	53
	1.1.0Beta2	63,799	587
TuxGuitar	0.1pre	11,209	122
	1.5.2	108,495	1,618

[9] http://freemind.sourceforge.net/wiki/index.php/Main_Page.
[10] http://www.jedit.org/.
[11] http://www.tuxguitar.com.ar.
[12] https://sourceforge.net.
[13] Download data points taken on August 8th, 2019.

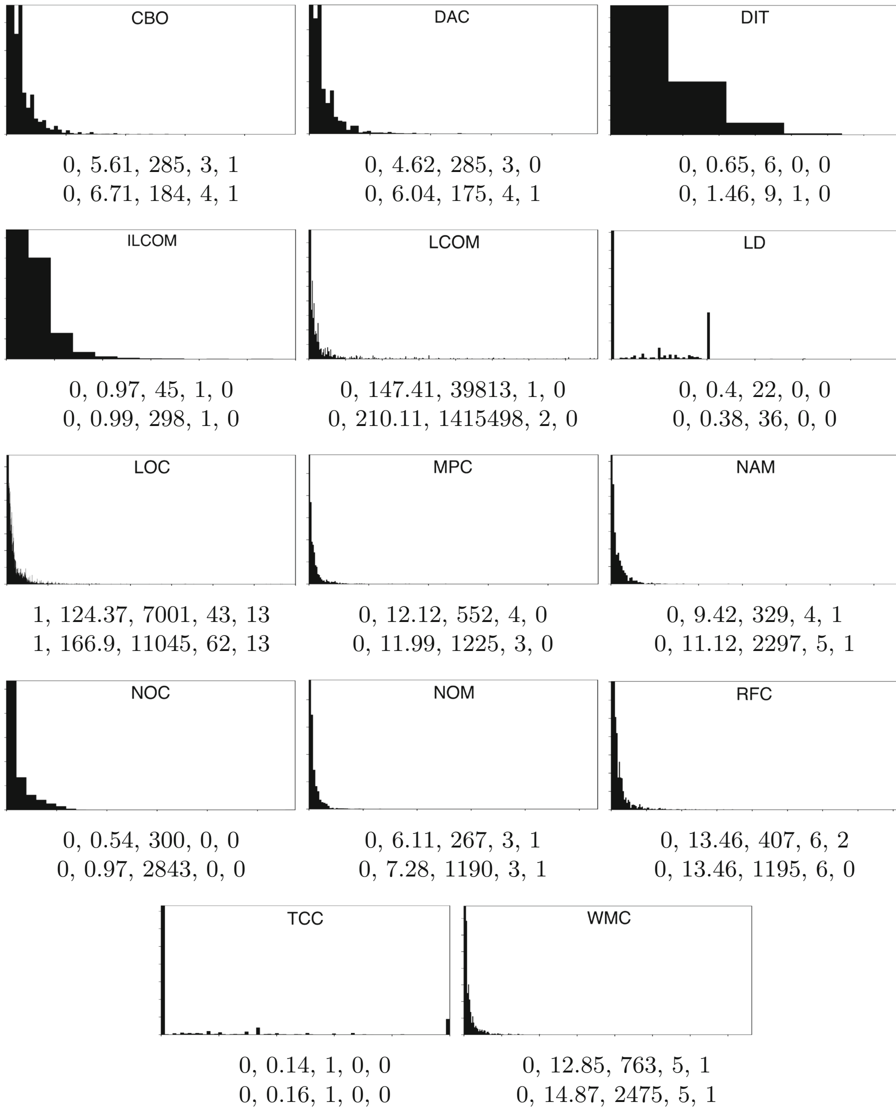

Fig. 2. Code metric histograms. Data labels: minimum, mean, maximum, median, modus. Our results on top row, results from [5] on bottom row for comparison (data from [26]).

As a preparatory step, each studied version was imported into an IDE. We ensured that library source code was separated from actual application code in order to not affect our analysis. Since we employed Java 8, we encountered compilation errors with older versions of the applications that were developed using earlier versions of the Java platform. The issues were resolved taking into

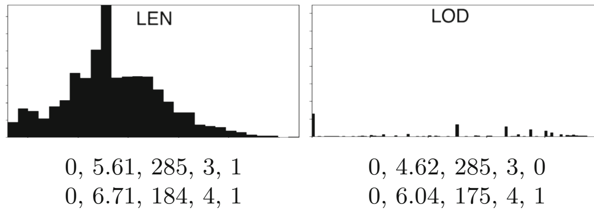

Fig. 3. Documentation metric histograms. Data labels: minimum, mean, maximum, median, modus. Our results on top row, results from [5] on bottom row for comparison.

account not to alter the results of metric extraction. We assured that for each application, all mandatory source code was included, testing all available functionalities in detail. The raw metric data that was extracted is available on our website[14]. Using this data, we developed a number of scripts in order to extract only the required metric values for our study for each application version as well as in aggregate form.

Data collection was helped by the fact that for each application, its complete development history was available on SourceForge. Furthermore, released versions were clearly marked, dated and had associated binaries and source code. In total, we included 38 versions of FreeMind, 43 for jEdit and 26 for TuxGuitar.

Table 2. Mean and median metric values per application.

	FreeMind	jEdit	TuxGuitar	[5]	FreeMind	jEdit	TuxGuitar	[5]
CBO	5.36	4.67	7.32	6.71	3.00	3.00	5.00	4.00
DAC	4.21	4.09	6.08	6.04	2.00	2.00	4.00	4.00
DIT	0.79	0.42	0.87	1.46	0.00	0.00	1.00	1.00
ILCOM	1.00	0.77	1.25	0.99	1.00	1.00	1.00	1.00
LCOM	197.62	124.83	130.81	210.11	2.00	1.00	2.00	2.00
LD	0.49	0.35	0.40	0.38	0.00	0.00	0.00	0.00
LEN	16.87	13.67	16.88	15.04	16.00	13.00	16.00	14.00
LOC	108.62	156.44	90.97	166.90	40.00	51.00	38.00	62.00
LOD	0.80	0.76	0.92	0.47	1.00	1.00	1.00	0.50
MPC	10.92	9.46	17.49	11.99	4.00	3.00	5.00	3.00
NAM	9.75	8.41	10.67	11.12	4.00	3.00	5.00	5.00
NOC	0.65	0.37	0.71	0.97	0.00	0.00	0.00	0.00
NOM	6.88	5.16	6.80	7.28	3.00	2.00	3.00	3.00
RFC	13.54	10.62	17.78	13.46	6.00	5.00	8.00	6.00
TCC	0.14	0.15	0.16	0.16	0.00	0.00	0.00	0.00
WMC	12.51	13.40	12.36	14.87	5.00	5.00	4.00	5.00
	Mean values ([26])				Median values			

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

4.2 Quantitative Statistics

In this section we provide an initial overview of the extracted metric values, and compare them with the results presented in [5]. For each of the target applications, we create its own data set, comprising metric values extracted from all studied versions of that application. This enables statistical comparison across applications in order to identify any existing trends. The data from all 107 application versions is coalesced into an aggregated data set. We compare the aggregated data against the results reported in [5], where authors carried out a cross-sectional study of 146 open-source Java applications.

Given the large number of data points recorded for our study[15], we detail those aspects that were found of most interest. We remind the interested reader that the entire metric data set is freely available on our website.

Histograms for code and documentation metric values in our aggregated data set are shown in Figs. 2 and 3. They also provide a faithful representation of the value distributions from the three target application data sets. This also holds when comparing our data with that presented in [5]. We find that histograms are similar even in the case of metrics having stand-out values, such as LD, LOD and TCC, where the value of 1 is frequent[16]. LEN appears to be the only metric with normal distribution.

Descriptive statistics for every metric in the aggregated data set, as well as corresponding ones from [5] are shown below the histograms in Figs. 2 and 3. We notice that in every case, the smallest recorded values are the minimal ones, which is 0 for all metrics with the exception of LOC, where it is 1. Maximal values are outliers and show much more variance, both across studied application versions and across the data sets. As such, our study will focus mostly around median and mean metric values, and detail extreme values only where it makes sense.

Examination of the mean, median and modus values proves to be of much more interest. Our first observation is that median and modus values are close across all the five data sets, for each of the 16 studied metrics. This is detailed in Table 2, where mean and median values for each application data set, as well as those recorded by Barkmann et al. [5] are shown. When examining these values, one must also consider the range for each metric, as detailed in Sect. 2. We observe that for CBO, NAM, NOM, TCC and WMC mean values are close across the data sets. Values for LEN and LOD show that while in most cases, the length of used identifiers is suitable, open-source applications appear to lack inline documentation. This is especially true in the case of our target applications, where more than 80% of methods remain undocumented. The data also illustrates the existance of application-specific trends. We observe that jEdit classes tend to be larger, as illustrated by higher LOC than FreeMind and TuxGuitar, being very close to the mean LOC reported in [5]. At the same time, jEdit shows a more flat inheritance hierarchy, illustrated by lower DIT and NOC values when compared

[15] 107 application versions x 16 studied metrics x 5 data points = 8,560 data points.

[16] In the case of TCC 1 is the maximal value.

Table 3. Metric dependencies in FreeMind (top row), jEdit (second row), TuxGuitar (third row) and as reported in [5] (bottom row). LEN and LOD metrics omitted as no strong dependencies were found. Data from [26].

Metric	CBO	DAC	DIT	ILCOM	LCOM	LD	LOC	MPC	NAM	NOC	NOM	RFC	TCC	WMC
DAC	**0.97** **0.98** **0.96** **0.98**	1.00												
DIT	0.28 0.18 0.18 0.52	0.30 0.20 0.10 0.52	1.00											
ILCOM	0.46 0.44 0.07 0.53	0.49 0.46 0.11 0.41	0.08 −0.00 −0.29 0.39	1.00										
LCOM	0.53 0.55 0.20 0.53	0.56 0.56 0.21 0.55	0.05 −0.03 −0.12 0.40	0.55 0.40 0.37 0.47	1.00									
LD	0.20 0.18 0.03 0.31	0.22 0.21 0.06 0.33	0.07 0.15 −0.20 0.43	0.40 0.56 0.43 0.79	0.11 0.07 0.11 0.44	1.00								
LOC	0.58 0.77 0.46 0.58	0.61 0.78 0.46 0.60	0.09 −0.00 −0.14 0.14	0.56 0.55 0.34 0.47	0.77 **0.84** 0.66 0.58	0.25 0.21 0.16 0.32	1.00							
MPC	**0.83** **0.83** 0.62 **0.83**	**0.81** **0.82** 0.56 **0.81**	0.22 0.06 0.03 0.53	0.46 0.44 0.18 0.57	0.60 0.75 0.56 0.59	0.17 0.15 0.04 0.50	0.66 **0.87** **0.82** 0.66	1.00						
NAM	0.69 0.71 0.30 0.51	0.72 0.72 0.30 0.53	0.11 −0.01 −0.23 0.16	0.72 0.65 0.57 0.63	**0.86** **0.85** 0.78 0.68	0.32 0.29 0.29 0.46	**0.85** **0.94** 0.78 **0.83**	0.71 **0.82** 0.59 0.62	1.00					
NOC	−0.01 −0.04 −0.02 0.06	0.02 −0.03 −0.03 0.08	−0.03 −0.05 −0.06 0.40	0.10 0.02 0.02 0.57	0.14 0.01 0.01 0.38	0.01 −0.01 0.02 0.62	0.06 0.01 −0.02 −0.11	0.02 −0.02 −0.02 0.21	0.13 0.01 0.01 0.06	1.00				
NOM	0.56 0.68 0.32 0.56	0.60 0.69 0.33 0.58	0.10 −0.05 −0.23 0.23	0.65 0.59 0.55 0.59	**0.91** **0.90** **0.83** 0.79	0.23 0.20 0.27 0.48	**0.82** **0.94** **0.83** 0.79	0.63 **0.84** 0.67 0.65	**0.95** **0.96** **0.92** **0.91**	0.16 0.03 0.03 0.14	1.00			
RFC	0.74 **0.83** 0.53 0.71	0.74 **0.82** 0.49 0.70	0.18 0.02 −0.02 0.27	0.62 0.53 0.32 0.52	**0.84** **0.82** 0.62 0.71	0.23 0.18 0.12 0.01	**0.80** **0.92** **0.88** **0.80**	**0.88** **0.96** **0.92** **0.81**	**0.91** **0.91** 0.73 **0.83**	0.11 −0.01 −0.01 0.02	**0.90** **0.93** **0.82** **0.90**	1.00		
TCC	0.02 0.05 0.08 0.33	0.02 0.07 0.09 0.35	0.02 0.05 −0.05 0.54	0.11 0.25 0.04 0.78	−0.04 −0.01 −0.05 0.46	0.22 0.43 0.25 **0.80**	0.03 0.09 0.03 0.26	0.04 0.06 −0.01 0.51	0.05 0.12 0.07 0.41	−0.02 −0.04 −0.05 **0.84**	0.02 0.08 0.02 0.45	0.04 0.08 0.01 0.36	1.00	
WMC	0.53 0.70 0.38 0.59	0.55 0.70 0.38 0.60	0.08 −0.04 −0.17 0.20	0.61 0.53 0.37 0.57	**0.86** **0.88** 0.72 0.72	0.23 0.16 0.16 0.44	**0.89** **0.95** **0.95** **0.84**	0.69 **0.87** **0.82** 0.71	**0.90** **0.93** 0.79 **0.88**	0.12 0.01 −0.01 0.05	**0.93** **0.96** **0.88** **0.93**	**0.90** **0.93** **0.88** **0.93**	0.04 0.08 0.01 0.40	1.00

to the other applications. As a matter of fact, our studied applications tend to have shallower inheritance trees than those from [5].

4.3 Metric Dependencies

Several metric value-based characterizations of software have been proposed in existing literature. However, many of them eschew a thorough study of the relations between numerical metric values. We believe that understanding existing correlations between metrics can further assist researchers in proposing and evaluating metric-based models. In this section we identify existing metric dependencies in the target applications and cross-check our data against [5].

As shown in Figs. 2 and 3, LEN is the only metric having a normal distribution. This, together with the difference in metric value ranges shown in Sect. 2, determined us to employ Spearman's rank correlation to determine metric dependency. Correlation data per application, including results from [5] are shown in Table 3. We establish a threshold of 0.8 in absolute value for *strong* correlations, which are highlighted and discussed below. In order to keep Table 3 readable, we did not include the LEN and LOD metrics, both of which appeared to be independent from other metrics as well as each other. The only exception is a weak correlation between DIT and LEN, which appeared in all studied applications, as well as [5]. It is explained by the tendency of derived classes in inheritance hierarchies to have more detailed names than those of base classes or interfaces.

Metric correlations in our target applications follow the trends identified by Barkman et al. [5]. We examine our results through the lens of the four characteristics of object-oriented software presented in Sect. 2.

We observe that strong and consistent correlations exist between coupling metrics CBO, DAC and MPC, as well as size-related metrics LOC, NAM and NOM. This was expected, as an increase in attributes or method count leads to increased class sizes when measured using metrics that predate object orientation. The same explanation covers the strong observed correlation between structural complexity RFC and WMC.

The NOM metric is also correlated with LCOM and NAM. This confirms that an increased method count usually leads to a lack of cohesion. As the number of class methods is a part of the NAM metric, this correlation was also expected. Inheritance metrics DIT and NOC remain uncorrelated in all data sets, challenging the expectation that classes at the base of the inheritance tree have more children.

An interesting result is that cohesion metrics LCOM, ILCOM and TCC do not show strong correlation in either of the studied data sets. LCOM shows a weak correlation with its improved variant in all data sets, showing that while they measure similar software aspects, there is enough differentiation between them. The result for TCC is more interesting, as the cross-sectional study in [5] showed much stronger correlation than observed by us. We believe this is a result of target application selection, which highlights the necessity of backing up any metric-based model with exploratory evaluation.

Table 4. Metric dependencies in FreeMind (top row - below Q1, middle row - interquartile range, bottom row - above Q3).

Metric	CBO	DAC	DIT	ILCOM	LCOM	LD	MPC	NAM	NOC	NOM	RFC	TCC	WMC
DAC	0.68 **0.83** **0.99**	1.00											
DIT	0.34 0.42 0.26	0.59 0.55 0.25	1.00										
ILCOM	0.04 0.04 0.37	0.00 0.17 0.40	−0.10 −0.03 0.00	1.00									
LCOM	−0.10 0.08 0.49	−0.05 0.20 0.51	−0.12 0.07 −0.01	−0.20 −0.01 0.55	1.00								
LD	0.09 0.16 0.04	0.04 0.25 0.06	−0.07 0.08 −0.08	**0.88** 0.66 0.18	−0.16 −0.07 0.02	1.00							
MPC	0.67 0.78 **0.81**	0.21 0.59 0.79	−0.07 0.32 0.20	0.30 −0.03 0.36	−0.13 −0.05 0.56	0.36 0.14 −0.01	1.00						
NAM	−0.12 0.13 0.65	−0.12 0.32 0.68	−0.29 0.10 −0.01	0.37 0.53 0.66	0.69 0.64 **0.89**	0.27 0.41 0.18	−0.03 0.03 0.65	1.00					
NOC	−0.19 −0.10 0.02	−0.04 −0.04 0.02	−0.11 −0.08 0.08	−0.11 −0.03 0.22	0.33 0.22 0.21	−0.10 −0.03 0.04	−0.26 −0.17 0.05	0.26 0.19 0.18	1.00				
NOM	−0.08 0.16 0.48	−0.09 0.32 0.50	−0.25 0.14 −0.04	0.07 0.27 0.59	**0.84** **0.80** **0.95**	0.00 0.21 0.07	−0.07 0.04 0.55	**0.88** **0.90** **0.94**	0.33 0.24 0.23	1.00			
RFC	0.50 0.66 0.68	0.07 0.61 0.68	−0.24 0.29 0.09	0.23 0.15 0.54	0.44 0.45 **0.86**	0.24 0.25 0.04	0.71 0.74 **0.85**	0.53 0.58 **0.88**	0.02 0.06 0.18	0.57 0.63 **0.88**	1.00		
TCC	−0.13 −0.01 −0.10	−0.07 0.07 −0.12	0.00 0.01 −0.10	0.48 0.31 −0.15	−0.11 −0.14 −0.18	0.27 0.35 0.07	−0.12 0.05 −0.09	0.32 0.27 −0.17	−0.03 0.00 −0.03	0.18 0.14 −0.20	0.00 0.16 −0.16	1.00	
WMC	0.09 0.23 0.42	−0.01 0.33 0.43	−0.23 0.09 −0.06	0.04 0.22 0.53	0.76 0.59 **0.88**	0.01 0.24 0.06	0.08 0.23 0.60	0.78 0.77 **0.86**	0.27 0.13 0.18	**0.90** **0.83** **0.91**	0.65 0.70 **0.87**	0.12 0.18 −0.13	1.00

4.4 The Confounding Effect of Class Size

The confounding effect class size has on metric value-based measurements was reported by El Emam et al. [12]. Due to its significance, class size must be accounted for when studying metric dependencies. Authors of [12] showed that in many cases, metric dependencies could be explained by larger classes having higher metric values, which confounds data interpretation. As shown in Table 3, the LOC metric appears correlated with most of the metrics. The exceptions are DIT, LEN, LOD, NOC and TCC, which do not exhibit correlation with LOC, or other metrics.

Table 5. Metric dependencies in jEdit (top row - below Q1, middle row - inter-quartile range, bottom row - above Q3).

Metric	CBO	DAC	DIT	ILCOM	LCOM	LD	MPC	NAM	NOC	NOM	RFC	TCC	WMC
DAC	**0.82**												
	0.94	1.00											
	0.99												
DIT	0.34	0.50											
	0.33	0.39	1.00										
	0.09	0.09											
ILCOM	−0.06	0.01	0.05										
	−0.08	−0.02	−0.02	1.00									
	0.36	0.36	−0.13										
LCOM	0.07	0.08	−0.06	−0.12									
	−0.04	−0.02	−0.02	0.11	1.00								
	0.61	0.62	−0.09	0.47									
LD	−0.10	−0.02	0.04	0.78	−0.10								
	−0.11	−0.04	0.11	0.66	−0.03	1.00							
	−0.02	−0.01	0.16	0.23	0.00								
MPC	0.68	0.45	0.11	−0.09	0.01	−0.08							
	0.74	0.66	0.28	−0.03	−0.04	0.01	1.00						
	0.84	**0.83**	0.01	0.40	0.78	−0.03							
NAM	−0.15	−0.05	−0.21	0.40	0.39	0.38	−0.22						
	−0.10	−0.02	0.00	0.62	0.34	0.49	−0.01	1.00					
	0.70	0.70	−0.09	0.59	**0.91**	0.07	**0.82**						
NOC	−0.17	−0.08	−0.11	−0.08	0.16	−0.06	−0.21	0.04					
	−0.11	−0.07	−0.07	−0.04	0.11	−0.01	−0.14	0.00	1.00				
	−0.07	−0.06	−0.01	0.07	0.02	0.01	−0.04	0.00					
NOM	0.15	0.10	−0.09	0.06	**0.88**	0.03	0.09	0.33	0.17				
	−0.06	−0.02	−0.01	0.46	0.70	0.34	−0.02	0.65	0.08	1.00			
	0.68	0.68	−0.17	0.56	**0.94**	0.01	**0.83**	**0.97**	0.02				
RFC	0.56	0.30	−0.05	−0.05	0.44	−0.06	0.76	0.03	−0.06	0.60			
	0.59	0.52	0.21	0.15	0.26	0.14	**0.83**	0.22	−0.09	0.41	1.00		
	0.83	**0.82**	−0.07	0.49	**0.86**	−0.02	**0.96**	**0.92**	−0.02	**0.93**			
TCC	−0.07	−0.07	−0.03	0.21	−0.04	0.08	−0.04	0.15	−0.02	0.08	0.02		
	−0.08	−0.06	0.06	0.33	−0.13	0.44	−0.02	0.31	−0.05	0.26	0.07	1.00	
	−0.14	−0.15	−0.06	−0.09	−0.08	0.10	−0.07	−0.09	−0.08	−0.10	−0.09		
WMC	0.28	0.07	−0.19	−0.07	0.52	−0.09	0.40	0.08	0.00	0.67	0.71	0.03	
	0.27	0.21	−0.10	0.18	0.32	0.08	0.39	0.30	−0.08	0.55	0.63	0.13	1.00
	0.69	0.69	−0.14	0.49	**0.92**	−0.05	**0.87**	**0.93**	−0.01	**0.96**	**0.94**	−0.08	

To determine the effect class size has on metric dependencies, we partitioned all analyzed classes into quartiles using the LOC metric. We calculated the metric dependencies for each of our three data sets below the first quartile (below Q1), between the quartiles, and above the third quartile (above Q3). The detailed result is illustrated per application in Tables 4, 5 and 6. The LOC metric itself was omitted, as we had already used it to partition the data.

Table 6. Metric dependencies in TuxGuitar (top row - below Q1, middle row - inter-quartile range, bottom row - above Q3).

Metric	CBO	DAC	DIT	ILCOM	LCOM	LD	MPC	NAM	NOC	NOM	RFC	TCC	WMC
DAC	**0.92**												
	0.92	1.00											
	0.98												
DIT	0.64	0.60											
	0.60	0.53	1.00										
	0.10	0.05											
ILCOM	−0.26	−0.23	−0.32										
	−0.39	−0.32	−0.43	1.00									
	0.08	0.08	−0.15										
LCOM	−0.07	−0.01	−0.12	−0.12									
	−0.17	−0.15	−0.19	0.49	1.00								
	0.12	0.11	−0.14	0.41									
LD	−0.07	−0.04	−0.24	0.41	−0.11								
	−0.16	−0.07	−0.22	0.36	0.15	1.00							
	−0.08	−0.06	−0.18	0.42	0.06								
MPC	**0.82**	0.66	0.58	−0.25	−0.21	−0.02							
	0.77	0.60	0.44	−0.35	−0.14	−0.18	1.00						
	0.49	0.43	0.09	0.17	0.54	−0.13							
NAM	−0.10	−0.05	−0.20	0.30	0.66	0.09	−0.20						
	−0.19	−0.11	−0.30	0.60	0.52	0.30	−0.21	1.00					
	0.10	0.08	−0.18	0.51	**0.84**	0.15	0.52						
NOC	−0.16	−0.13	−0.16	−0.10	0.11	−0.07	−0.18	0.04					
	0.01	−0.02	−0.06	0.07	0.31	0.05	0.01	0.13	1.00				
	−0.06	−0.05	−0.08	0.27	0.12	0.20	−0.03	0.15					
NOM	−0.05	0.03	−0.17	0.12	**0.90**	−0.03	−0.22	0.75	0.13				
	−0.23	−0.17	−0.34	0.64	0.77	0.33	−0.15	0.70	0.21	1.00			
	0.16	0.14	−0.18	0.50	**0.88**	0.14	0.61	**0.92**	0.15				
RFC	0.79	0.66	0.47	−0.21	0.26	−0.10	**0.82**	0.18	−0.10	0.30			
	0.63	0.46	0.28	−0.03	0.26	−0.05	**0.83**	0.10	0.13	0.29	1.00		
	0.35	0.30	0.06	0.28	0.61	−0.06	**0.90**	0.65	0.03	0.77			
TCC	−0.14	−0.08	−0.20	0.24	−0.08	0.48	−0.12	0.12	−0.06	0.11	−0.10		
	0.05	0.13	−0.05	0.05	−0.18	0.32	−0.03	0.19	−0.05	0.13	−0.02	1.00	
	0.02	0.02	0.02	−0.14	−0.15	0.02	−0.11	−0.11	−0.09	−0.18	−0.14		
WMC	0.04	0.10	−0.13	0.09	**0.86**	0.00	−0.12	0.69	0.10	**0.95**	0.38	0.14	
	−0.12	−0.07	−0.34	0.45	0.59	0.32	0.04	0.54	0.17	**0.83**	0.40	0.13	1.00
	0.23	0.21	−0.10	0.29	0.72	−0.06	0.79	0.72	0.00	**0.84**	**0.85**	−0.17	

Immediately we observe that most of the strong metric dependencies occur in classes above the third quartile, which confirms El Emam et al.'s observation of the important role played by class size in metric dependencies. LCOM, NAM and RFC appear sensitive to class size across all target applications, showing strong dependencies for classes above Q3. An inverse relation is observed between DIT on one hand, and CBO and DAC on the other. In this case, we notice

Table 7. Extreme values for metric means for early (left) and *mature* application versions (right). Includes data from [26].

	FreeMind				jEdit				TuxGuitar			
	<1.0.0Alpha4		≥1.0.0Alpha4		<4.0pre4		≥4.0pre4		<1.0rc1		≥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	4.29	4.29	4.91	6.03	7.56	7.06	7.88
DAC	2.67	5.30	4.20	4.38	3.45	3.83	3.77	4.30	4.76	5.46	5.16	6.97
DIT	0.15	1.69	0.70	1.03	0.37	0.70	0.32	0.43	0.45	0.55	0.78	1.07
ILCOM	0.81	1.04	0.99	1.04	0.49	0.79	0.79	0.83	1.07	1.33	1.15	1.46
LCOM	84.85	193.25	196.85	237.90	43.44	117.75	126.79	149.31	90.94	130.49	117.15	176.79
LD	0.30	0.52	0.48	0.51	0.23	0.36	0.34	0.37	0.39	0.48	0.35	0.50
LEN	11.77	17.07	16.67	17.17	12.25	13.10	13.01	14.35	14.84	15.09	15.19	18.26
LOC	63.35	157.84	100.05	110.79	91.29	153.94	158.64	177.37	94.93	116.69	73.13	115.25
LOD	0.72	0.91	0.78	0.81	0.73	0.82	0.73	0.80	0.68	0.83	0.88	0.99
MPC	6.99	13.20	10.59	10.92	6.79	9.00	9.34	10.05	14.26	21.27	14.65	22.85
NAM	7.06	9.84	9.85	10.09	5.18	9.02	8.53	9.19	9.71	12.13	9.41	12.98
NOC	0.15	1.44	0.59	0.63	0.31	0.65	0.29	0.38	0.45	0.52	0.58	0.92
NOM	5.26	7.06	6.88	6.99	3.16	5.49	5.28	5.46	6.38	7.23	6.13	8.13
RFC	9.74	15.17	13.49	13.62	7.91	10.39	10.39	11.14	14.81	19.50	15.80	22.21
TCC	0.03	0.16	0.14	0.16	0.06	0.13	0.14	0.17	0.14	0.22	0.12	0.18
WMC	8.52	14.41	12.32	12.55	8.52	14.13	13.43	15.05	12.02	14.53	10.63	15.38

dependency strength decrease for larger class sizes. This is to be expected, as most metrics capture state and behaviour introduced by the class itself, disregarding inherited attributes. As such, many classes deep in inheritance hierarchies appear deceptively simple, as much of their complexity is hidden in base classes.

Even with class size accounted for, we still observe highly dependent metric pairs. Coupling metrics CBO and DAC, as well as complexity metrics NOM and WMC illustrate this best. In the same way, metric pairs that we observed to be independent in the previous section remain so even when partitioned according to class size. DIT, NOC and TCC showed no strong dependency in any of the data partitions.

4.5 Longitudinal Evaluation

This section is dedicated to an examination of the changes to metric values during application development. Data points illustrated in Figs. 2 and 3 are available for every metric and application version on our website. We found that values follow the illustrated distributions across all target application versions. As detailed in Sect. 4.2, maximum data points represent outliers, while minimal data points coincide with metric minimum values and are not interesting. As such, the present section is focused on discussing mean and median metric values. For the sake of brevity, we do not include all 8,560 data points. Our principle findings are that early application versions show more variability in metric values

Table 8. Application versions showing significant variance in metric values.

Application	Version	LOC	Classes
jEdit	2.6final	46,671	453
	3.0final	40,756	282
FreeMind	0.7.1	18,928	199
	0.8.0	84,199	718
	0.8.1	84,089	718
	0.9.0Beta17	56,752	577
TuxGuitar	1.2	77,056	736
	1.3.0	91,481	1,234

and that key application versions can be identified during which large changes to metric values occur.

Metric Variability in Early and Mature Versions. We examined the changes to metric values that occurred between consecutive versions of the same application. For all three target applications, we found that some of the most consistent changes occurred within early releases of the application. Of course, there exists no structured definition for an *"early version"*, especially not one that can be used across several applications. As such, we used our familiarity with the studied applications to identify the earliest version that we considered *mature*. In the case of our target applications, they were FreeMind 1.0.0Alpha4, jEdit 4.0pre4 and TuxGuitar 1.0rc1. These versions include most of the functionalities available in the latest version of the respective application, have the same look & feel as all subsequent versions and appear to be stable software releases. Table 7 illustrates minimum and maximum mean metric values in both early and *mature* application versions.

We observe that for all applications, metric variability is much higher for the earlier versions. As shown in Table 1, the first version of FreeMind consisted of 3,722 lines of code, fewer than the first version of TuxGuitar (11,209). In contrast, the first release of jEdit (33,768 LOC) was much more mature, and already contained the application's most important functionalities. On the other hand, once the application architecture is established and the principal functionalities set is implemented, we observe a significant reduction in the variability of metric values between versions. This is illustrated for each application, in the right-hand columns of Table 7. Furthermore, longitudinal examination also showed that specific trends can be identified for each application with regards to how object-oriented concepts such as coupling, inheritance and structural complexity are handled. It is our opinion that additional case studies presenting a longitudinal view are required before desirable metric ranges and most importantly, reliable metric-based characterisations can be established.

Table 9. Mean metric values for given application versions.

	FreeMind				jEdit		TuxGuitar	
	0.7.1	0.8.0	0.8.1	0.9.0Beta17	2.6final	3.0final	1.2	1.3.0
CBO	4.75	6.15	6.15	5.31	4.24	4.29	7.05	7.07
DAC	3.10	5.29	5.29	4.14	3.73	3.82	5.22	6.30
DIT	0.50	1.69	1.69	0.74	0.63	0.42	0.79	0.95
ILCOM	0.95	0.80	0.80	1.03	0.52	0.77	1.43	1.22
LCOM	179.54	152.56	152.56	189.32	47.95	114.37	176.79	130.11
LD	0.42	0.43	0.43	0.51	0.25	0.36	0.50	0.35
LEN	15.23	16.91	16.91	17.06	12.62	12.97	15.45	18.26
LOC	102.95	157.83	157.52	97.87	100.16	151.28	115.24	80.74
LOD	0.86	0.72	0.72	0.80	0.82	0.73	0.89	0.98
MPC	11.53	13.19	13.19	10.51	7.58	9.00	22.82	14.64
NAM	9.08	8.99	8.99	9.77	6.01	9.01	12.96	9.85
NOC	0.37	1.44	1.44	0.61	0.56	0.33	0.65	0.64
NOM	6.61	7.06	7.06	6.82	3.72	5.48	8.12	6.31
RFC	13.07	15.16	15.16	13.29	8.84	10.38	22.20	15.80
TCC	0.06	0.08	0.08	0.15	0.08	0.12	0.16	0.12
WMC	12.94	14.40	14.38	12.16	9.27	14.00	15.38	11.30

Causes of Large Variations in Metric Values. We also observed that metric values were consistent between most consecutive version pairs of the studied applications. At the same time, we could identify version pairs where metric values were greatly disrupted. We illustrate these pairs using Table 8. The table also includes information about LOC and the number of classes, in order to help understand the causes behind observed variations. For example, it is obvious that a large push in development between FreeMind 0.7.1 and 0.8.0 contributed to significant changes to metric values, as evidenced by the sharp increase in application LOC and class count. The same can be said about TuxGuitar version 1.3.0. The opposite however is true for jEdit 3.0final, as well as FreeMind 0.9.0Beta17. In these versions we observe important decreases in both LOC and class count, most likely a result due to refactoring.

Table 9 illustrates mean metric values for the highlighted application versions. For each version, we manually examined its source code in detail to identify the underlying changes leading to these variations.

FreeMind 0.8.0 contains major changes, as already evidenced by the sharp increase in LOC and class count. It is the first version to use external libraries for XML processing and input forms. During use, it is clear that FreeMind 0.8.0 is more complex and fully-featured, with many changes that are visible at UI level, including more complex application preferences and features for mind map and node management. Its scope remains apparent at source file level, with only 21 out of the 92 source files remaining unchanged from 0.7.1. The number of source files also increased greatly in the newer version, from 92 to 469. Much

of the observed discrepancy between numbers of source files, classes and LOC between the versions can be explained by the newer application including 272 classes that were generated by the JAXB libraries encoding most of the actions that can be performed using the application. These classes contributed with 49,434 lines to the inflation of LOC witnessed between the studied versions. Between version 0.8.0 and 0.8.1, no source files were added or deleted, but many of them have undergone small updates. This includes all generated code, that was regenerated for version 0.8.1. FreeMind again underwent significant changes for version 0.9.0Beta17, an evolution from 0.8.1. Out of 469 source files in version 0.8.1, only 127 can be found in the newer version, and all of them have undergone changes. Version 0.9.0Beta17 also added 230 new Java source files, covering all functionality areas. Action source files generated using JAXB in version 0.8.0 were replaced with a smaller number of hand-written classes with similar naming and functionality. This explains most of the class count and LOC difference between versions 0.8.1 and 0.9.0Beta17.

In the case of jEdit, version 3.0final was the only one where mean metric values were disrupted. A possible contributor to this is that relatively, early analyzed versions were more mature than equivalent ones from the other applications. In the case of version 3.0final, we observed that the package *"org.gjt.sp.jedit.actions"*, which contained 153 event handler classes with low statement count and cyclomatic complexity was deleted. These were replaced with an XML file that provides action descriptors together with Java-like code snippets that are executed when the action is fired. Only 81 source files out of 341 remained unchanged between these versions.

In the case of TuxGuitar version 1.3.0, the *"org.herac.tuxguitar.gui"* package was split into **.app*, **.editor* and **.graphics* packages. Most packages were updated or refactored. New plugins were added, existing ones have seen source code changes. Only 62 out of the 650 source code files remained unedited between these versions. Version 1.3.0 introduced 930 new source files, most of which contain code for custom application actions in the form of small classes having low complexity, skewing the mean and median metric values.

The last observation is related to the expectation that mean metric values increase in more advanced application versions. Our data showed this to be true mostly in the case of FreeMind and jEdit, especially in the case of size metrics LOC, NAM and NOM. However, as we have shown in this section, this is alleviated by the refactorings that were carried out in some of the versions.

Our examination resulted in several conclusions. First, we observed that most of the significant metric variations occurred in early application versions. This was true both as highlighted in Table 9, as well as when manually identifying versions with significant metric variations. In addition, we feel that a more in-depth discussion is warranted regarding the effect that large numbers of small, relatively straightforward classes have on software quality characteristics. The importance and magnitude these classes should have when building metric-based models has yet to be clarified. In several cases, we observed Java source code being replaced with XML descriptors. This is an illustrative example of the

inherent limitations of metric extraction tools and understanding of software based on metric values.

4.6 Threats to Validity

We carried out our study using the following steps, in order: preparing application versions, extracting metric data, processing the metric data and analysing it. We presented all the steps required to duplicate our study in detail. Extracted metric information, as well as aggregated data used for analysis is available on our website. Each target application version was manually examined in order to ensure that no factors that could influence metric values were present. We provided structured definitions for all metrics used, and extracted the data using a freely-available, cross-platform tool.

We selected three similar applications from a programming language and architecture standpoint. This helps limit external threats to validity related to application selection and generalization of results. This also allows comparing obtained results, as all three applications include the same layers. Application selection and metric extraction were finalized before data analysis, to eliminate selection bias. All results are presented both individually, per-application, as well as in aggregate form.

However, we believe one of our most important contributions was the comparative evaluation against a large-scale cross-sectional study that was carried out using the same methodology as ours. We believe this will help create a solid basis for additional studies towards a metric-based understanding of software quality and the software development process.

Among existing threats, we must include the limited number and types of studied applications. This means that additional research is required in order to draw conclusions about other types of software, such as non GUI-driven or mobile applications. Furthermore, as we only included open-source software, they might not be representative for other applications. As such, we believe that additional experimental evaluation is required in order to cover additional applications, programming languages as well as considered metrics.

5 Conclusions and Future Work

In this paper we establish a number of metrics that previous research has associated with software product quality. We select three open-source, user interface-driven applications developed in Java and analyze the values and relations between these metrics within each application's entire development history.

Each step of our evaluation is detailed and we employ open-source tooling to ensure that our evaluation is repeatable. At each step, we compare our results with a comparable large-scale evaluation, obtaining results from an aggregate of over 250[17] application versions. We believe these combined results provide a sound foundation to be used in further research.

[17] [5] evaluated 146 software projects.

We found that metric distributions, mean, median and modus values were consistent across the studies. Mean and median values prove stable once applications reach maturity, as evidenced in all three target applications. Comparing values across studied applications revealed the existence of trends in metric values, driven by the architecture and design of the underlying application.

With regards to identified metric dependencies, we could identify metric pairs showing strong correlation across applications and application versions, as well as certain metrics that did not show correlation with any others. We further investigated the confounding effect of class size in order to confirm our findings.

Our longitudinal approach also revealed that across many application version we could not witness significant changes to aggregated metric values. Where such changes occurred, they were mostly driven by application development as well as refactoring, and were reflected in object-oriented metric values.

An important avenue for further research regards a finer grained analysis, in order to detect significant changes at package and class levels, not just those that are visible at aggregated level. Our evaluation should be extended in order to cover other application types, including mobile and non user interface-driven software. We believe this type of research can lay the foundation for identifying suitable metric thresholds that point toward good design practices. Another aspect regards the role played by the programming language itself, as it too plays an influence on metric values.

The end goal of this research is represented by a characterization of good design and development practices, where software metrics will have an important role for understanding and controlling the software development process.

References

1. Al Dallal, J., Briand, L.C.: An object-oriented high-level design-based class cohesion metric. Inf. Softw. Technol. **52**(12), 1346–1361 (2010). https://doi.org/10.1016/j.infsof.2010.08.006
2. ARISA Compendium - Understandability for Reuse:. http://www.arisa.se/compendium/node39.html#property:UnderstandabilityR (2018). Accessed Nov 2018
3. Arlt, S., Banerjee, I., Bertolini, C., Memon, A.M., Schaf, M.: Grey-box GUI testing: efficient generation of event sequences. CoRR abs/1205.4928 (2012)
4. Bakar, N.S.S.A., Boughton, C.V.: Validation of measurement tools to extract metrics from open source projects. In: 2012 IEEE Conference on Open Systems, pp. 1–6, October 2012. https://doi.org/10.1109/ICOS.2012.6417648
5. Barkmann, H., Lincke, R., Löwe, W.: Quantitative evaluation of software quality metrics in open-source projects. In: 2009 International Conference on Advanced Information Networking and Applications Workshops, pp. 1067–1072, May 2009. https://doi.org/10.1109/WAINA.2009.190
6. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996). https://doi.org/10.1109/32.544352
7. Du Bois B., Mens, T.: Describing the impact of refactoring on internal program quality. In: Proceedings of the 8th International Workshop on Evolution of Large-scale Industrial Software Applications, pp. 37–48 (2003)

8. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994). https://doi.org/10.1109/32.295895
9. Poshyvanyk, D., Marcus, A.: The conceptual coupling metrics for object-oriented systems. In: 2006 22nd IEEE International Conference on Software Maintenance, pp. 469–478 (2006). https://doi.org/10.1109/ICSM.2006.67
10. Dash, Y., Dubey, S.K., Rana, A.: Maintainability prediction of object oriented software system by using artificial neural network approach. Int. J. Soft Comput. Eng. **2**(2), 420–423 (2012)
11. DeMarco, T.: Controlling Software Projects: Management, Measurement, and Estimates. Prentice Hall PTR, Upper Saddle River (1986)
12. Emam, K.E., Benlarbi, S., Goel, N., Rai, S.N.: The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans. Softw. Eng. **27**(7), 630–650 (2001). https://doi.org/10.1109/32.935855
13. Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. **31**(10), 897–910 (2005). https://doi.org/10.1109/TSE.2005.112
14. Halstead, M.: Elements of Software Science. Elsevier North-Holland (1977)
15. Heitlager, I., Kuipers, T., Visser, J.: A practical model for measuring maintainability. In: IEEE Proceedings of 6th International Conference on the Quality of Information and Communications Technology, pp. 30–39 (2007)
16. Hitz, M., Montazeri, B.: Measuring coupling and cohesion in object-oriented systems. In: Proceedings of International Symposium on Applied Corporate Computing, pp. 25–27 (1995)
17. Kanellopoulos, Y., et al.: Code quality evaluation methodology using the ISO/IEC 9126 standard. Int. J. Softw. Eng. Appl. **1**(3), 17–36 (2010). https://doi.org/10.5121/ijsea.2010.1302
18. Landman, D., Serebrenik, A., Vinju, J.: 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 2014, pp. 221–230. IEEE Computer Society, Washington, USA (2014). https://doi.org/10.1109/ICSME.2014.44
19. Lenhard, J., Blom, M., Herold, S.: Exploring the suitability of source code metrics for indicating architectural inconsistencies. Softw. Qual. J. **27**, 241–274 (2018). https://doi.org/10.1007/s11219-018-9404-z
20. Li, W., Henry, S.: Maintenance metrics for the object oriented paradigm. In: 1993 Proceedings First International Software Metrics Symposium, pp. 52–60, May 1993. https://doi.org/10.1109/METRIC.1993.263801
21. Lincke, R., Lundberg, J., Löwe, W.: Comparing software metrics tools. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis - ISSTA 2008, pp. 131–142 (2008). https://doi.org/10.1145/1390630.1390648
22. Marinescu, R.: Measurement and quality in object-oriented design. In: 21st IEEE International Conference on Software Maintenance (ICSM 2005), pp. 701–704, September 2005. https://doi.org/10.1109/ICSM.2005.63
23. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE–2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837
24. Melo, W.L., Briand, L.C., Basili, V.R.: Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems. University of Maryland, Computer Science Department, Technical report (1995)
25. Molnar, A., Motogna, S.: 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 2017, pp. 88–93. ACM, New York (2017). https://doi.org/10.1145/3143434.3143447

26. Molnar, A., Neamţu, A., Motogna, S.: Longitudinal evaluation of software quality metrics in open-source applications. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE, vol. 1, pp. 80–91. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007725600800091
27. Ott, L., Bieman, J.M., Kang, B.K., Mehra, B.: Developing Measures of Class Cohesion for Object-Oriented Software. Department of Computer Science, Michigan Technological University, Technical report (1970)
28. Rodriguez, D., Harrison, R.: An overview of object-oriented design metrics (2001)
29. Motogna, S., Vescan, A., Serban, C., Tirban, P.: An approach to assess maintainability change. In: 2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pp. 1–6 (2016). https://doi.org/10.1109/AQTR.2016.7501279
30. Saraiva, J.: A roadmap for software maintainability measurement. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 1453–1455. IEEE Press, Piscataway (2013). http://dl.acm.org/citation.cfm?id=2486788.2487035
31. Sarker, M.: An overview of object oriented design metrics. Umeå University, Sweden (2005)
32. Silva, R., Costa, H.: 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 (2015). https://doi.org/10.1109/CLEI.2015.7359472
33. Tang, M.H., Kao, M.H., Chen, M.H.: An empirical study on object-oriented metrics. In: Proceedings of the 6th International Symposium on Software Metrics, METRICS 1999, p. 242. IEEE Computer Society, Washington (1999). http://dl.acm.org/citation.cfm?id=520792.823979
34. Wilking, D., Farooq Kahn, U., Kowalewski, S.: An empirical evaluation of refactoring. e-Inform. Softw. Eng. J. **1**, 27–42 (2007)
35. Xu, J., Ho, D., Capretz, L.F.: An empirical validation of object-oriented design metrics for fault prediction. J. Comput. Sci. (2008). https://doi.org/10.3844/jcssp.2008.571.577
36. Yuan, X., Memon, A.M.: Generating event sequence-based test cases using GUI run-time state feedback. IEEE Trans. Softw. Eng. **36**(1), 81–95 (2010). https://doi.org/10.1109/TSE.2009.68
37. ARISA Compendium:. http://www.arisa.se/compendium/ (2020). Accessed Jan 2020