Extracting Relative Thresholds for Source Code Metrics

Paloma Oliveira†*, Marco Tulio Valente†, and Fernando Paim Lima*
†Department of Computer Science, UFMG, Brazil
*Department of Computing, IFMG, Brazil
paloma.oliveira@ifmg.edu.br, mtov@dcc.ufmg.br, fernando.lima@ifmg.edu.br

***Abstract*—Establishing credible thresholds is a central challenge for promoting source code metrics as an effective instrument to control the internal quality of software systems. To address this challenge, we propose the concept of relative thresholds for evaluating metrics data following heavy-tailed distributions. The proposed thresholds are relative because they assume that metric thresholds should be followed by most source code entities, but that it is also natural to have a number of entities in the "long-tail" that do not follow the defined limits. In the paper, we describe an empirical method for extracting relative thresholds from real systems. We also report a study on applying this method in a corpus with 106 systems. Based on the results of this study, we argue that the proposed thresholds express a balance between real and idealized design practices.**

***Index Terms*—Source code metrics; Relative thresholds; Software quality; Software measurement.**

I. Introduction

Since the inception of the first programming languages we are witnessing the proposal of a variety of metrics to measure source code properties like size, complexity, cohesion and coupling [1, 2, 3]. However, metrics are rarely used to control in an effective way the quality of software products. To promote the use of metrics as an effective measurement instrument, it is essential to establish credible thresholds [4, 5, 6]. In this way, software quality managers can rely on metrics, for example, to certify new components or to monitor the degradation in quality that happens due to software aging.

Typically, metric thresholds are defined based on the personal experience of software quality experts. For example, industrial code standards for Java recommend that classes should have no more than 20 methods and that methods should have no more than 75 lines of code [7]. Recently, Alves et al. proposed a more transparent method to derive thresholds from benchmark data [4]. They illustrated the application of the method in a large software corpus and derived, for example, thresholds stating that methods with McCabe complexity above 14 should be considered as very-high risk.

However, it is well-known that source code metric values usually follow heavy-tailed distributions [8, 9]. Therefore, in most systems it is "natural" to have source code entities not following the proposed thresholds for several reasons, including complex requirements, performance optimizations, machine-generated code, etc. In the particular case of coupling for example, recent studies show that high coupling is never entirely eliminated from software design and that in fact some degree of high coupling might be quite reasonable [10].

Inspired by such findings, we claim in this paper that *absolute* thresholds should be complemented by a second piece of information, denoting the percentage of entities the upper limit should be applied to. More specifically, we propose the concept of *relative thresholds* for evaluating source code metrics, which have the following format:

$$\boxed{\text{p\% of the entities should have } M \leq k}$$

where M is a source code metric calculated for a given software entity (method, class, etc), k is the upper limit, and p is the minimal percentage of entities that should follow this upper limit. For example, a relative threshold can state that "85% of the methods should have McCabe $\leq$ 14". Essentially, this threshold expresses that high-risk methods may impact the quality of a system when they represent more than 15% of the whole population of methods.

Our central contribution in this paper is the proposal of an empirical method to derive relative thresholds based on a statistical analysis of a software corpus and attempting to balance two forces. First, the derived relative thresholds should reflect *real design rules*, widely followed by the systems in the considered corpus. Second, the derived relative thresholds should not be based on rather lenient upper limits. For example, a threshold stating that "95% of the classes should have less than 100 attributes" is probably satisfied by most systems, since it is based on a very high number of attributes. For this reason, relative thresholds should also reflect *idealized design rules*, based on widely accepted quality principles [3].

The paper starts by describing the method proposed to extract relative source code metric thresholds (Section II) and follows by illustrating its application in a small scenario (Section III). Next, we report an extensive study, where we used our method to derive relative thresholds from 106 Java-based systems (Sections IV-A to IV-C). In this study, we also evaluated the following variations regarding the use of the proposed method: (a) its application in a subcorpus of the original corpus including systems sharing a common functional domain (Sections IV-D); (b) a historical analysis where we retrospectively evaluated the derived thresholds in previous versions of a subset of the systems in the corpus (Section IV-E); (c) an inequality analysis, where we evaluated the dispersion of the metric values among

Accepted for publication by IEEE. © 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

$$ComplianceRate[p,k] = \frac{\mid \{\ S \in Corpus \mid p\%\ \textit{of the classes in S have M} \leq k\ \} \mid}{\mid Corpus \mid}$$

$$penalty_1[p,k] = \begin{cases} \dfrac{Min - ComplianceRate[p,k]}{Min} & \text{if } ComplianceRate[p,k] < Min \\ 0 & \text{otherwise} \end{cases}$$

$$penalty_2[k] = \begin{cases} \dfrac{k - MedianTail}{MedianTail} & \text{if } k > MedianTail \\ 0 & \text{otherwise} \end{cases}$$

$$ComplianceRatePenalty[p,k] = penalty_1[p,k] + penalty_2[k]$$

Fig. 1. *ComplianceRate* and *ComplianceRatePenalty* functions

the classes that respect the proposed thresholds (Section IV-F). In the paper, we also discuss the main properties and limitations of the proposed method (Section V). Finally, we present related work (Section VI) and the conclusion (Section VII).

II. Relative Thresholds

This section presents in details the proposed method to extract relative source code metric thresholds. An illustrative example of its usage is presented in the next section.

Goal: We target metric values that follow heavy-tailed distributions, when measured at the level of classes (although it is straightforward the application to other source code entities, like methods, packages, etc). Basically, the goal is to derive *relative thresholds* with the following format:

$$\boxed{p\% \text{ of the classes should have } M \leq k}$$

where M is a given source code metric and p is the minimal percentage of classes in each system that should respect the upper limit k. Therefore, this relative threshold tolerates $(100 - p)\%$ of classes with $M > k$.

Input: First, we assume that the values of p and k that characterize a relative threshold for a metric M should emerge from a curated set of systems, which we call our *Corpus*.

Furthermore, to derive the values of p and k the proposed method relies on two constants, Min and $Tail$, which are used to drive the method towards providing some quality confidence to the derived thresholds. More specifically, these constants are used to convey the notions of real and idealized design rules, respectively. On the other hand, we define that real design rules should be followed by at least $Min\%$ of the systems in the considered *Corpus* ($0 < Min \leq 100$). On the other hand, to express the notion of idealized design rules, we first consider that the values of M in the *Corpus* should follow a heavy-tailed distribution, which is common in the case of source code metrics [8, 9]. We also assume that the tail of the distribution starts at the $Tail$-th percentile of the values of M in each system S in the *Corpus* ($0 < Tail \leq 100$). In other words, since the distributions are heavy-tailed, we expect to have classes with very high-values for any metric M (e.g., classes with more than 100 attributes). Although such classes are "natural", they do not represent an "ideal" class.

Method: Figure 1 defines the functions used to calculate the parameters p and k that define the relative threshold for a given metric M. First, the function *ComplianceRate[p,k]* returns the percentage of systems in the *Corpus* that follows the relative threshold defined by the pair $[p, k]$. Moreover, the method aims to find the values of p and k that produce a *ComplianceRate* with a minimal *penalty*. More specifically, we penalize a compliance rate function in two situations:

- A *ComplianceRate[p,k]* less than $Min\%$ receives a penalty proportional to its distance to this value, as defined by function $penalty_1[p,k]$. As mentioned, the proposed thresholds should reflect design practices that are widely common in the *Corpus*. Therefore, this penalty formalizes this guideline, by fostering the selection of thresholds followed by at least $Min\%$ of the systems in the considered $Corpus$.
- As mentioned, we assume that in each system the classes with high values of M correspond to $Tail\%$ of the classes. Moreover, we define that $Tail[S]$ is an array with the $Tail$-th percentile of the values of M in each system S in the *Corpus*; and we call $MedianTail$ the median of the values in $Tail[S]$. We assume that $MedianTail$ is an idealized upper value for M, i.e., a value representing classes that, although present in most systems, have very high values of M. Therefore, a given $ComplianceRate[p,k]$ receives a second penalty proportional to the distance between k and $MedianTail$, as defined by function $penalty_2[k]$.

As defined in Figure 1, the final penalty of a given threshold is the sum of $penalty_1[p,k]$ and $penalty_2[k]$, as defined by function $ComplianceRatePenalty[p,k]$. Finally, the relative threshold is the one with the lowest $ComplianceRatePenalty[p,k]$. In case of ties, we select the result with the highest p and then the one with the lowest k.

III. Illustrative Example

This section illustrates our method by reporting the derivation of thresholds for the Number of Attributes (NOA) metric, considering the systems in the Qualitas Corpus [11]. In order to derive the relative threshold for NOA, we will consider the following parameters:

- $Min = 90\%$, i.e., we penalize thresholds that are not followed by at least 90% of the systems in the corpus.
- $Tail = 90\%$, i.e., we penalize thresholds whose upper limits are greater than the median of the 90th percentile, regarding the NOA values in each system.

Considering these parameters, Figure 2 plots the values of the *ComplianceRate* function, for different values of p and k. As expected, *ComplianceRate* is a monotonically increasing function, on the values of k. Moreover, as we increase p the function starts to present a slower growth.

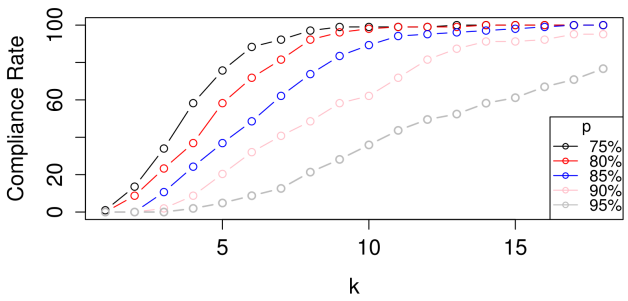


Fig. 2. Compliance Rate Function (NOA metric)

Figure 2 shows the importance of our second penalty. For example, we can check that $ComplianceRate[85, 17] = 100\%$, i.e., in 100% of the systems at least 85% of the classes have $NOA \leq 17$. However, in this case $MedianTail = 9$, i.e., the median of the 90th percentile for the NOA values in the considered systems is nine attributes. Therefore, the relative threshold defined by the pair $[85, 17]$ relies on a high value for k ($k = 17$) to achieve a compliance rate of 100%. To penalize a threshold like that, the value of $penalty_2$ is $(17-9)\ /\ 9 = 0.89$. Since $penalty_1 = 0$ (due to the 100% of compliance), we have that $ComplianceRatePenalty[85, 17] = 0.89$.

As can be observed in Figure 3, $ComplianceRatePenalty$ returns zero for the following pairs $[p, k]$:

$$[75, 7] \quad [75, 8] \quad [75, 9] \quad [\mathbf{80}, \mathbf{8}] \quad [80, 9]$$

Based on our tiebreaker criteria, we select the result with the highest p and the lowest k (in bold above), which leads to the following relative threshold:

80% of the classes should have $NOA \leq 8$

This threshold represents a balance between the two forces the method aims to balance. First, it reflects a *real design rule*,

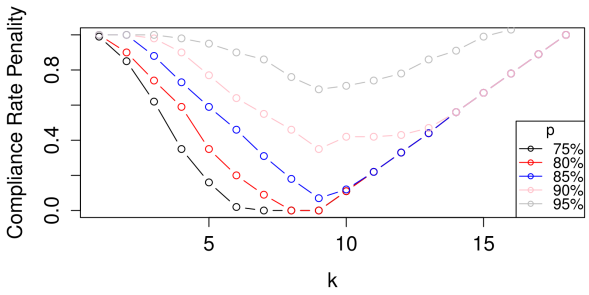


Fig. 3. Compliance Rate Penalty Function (NOA metric)

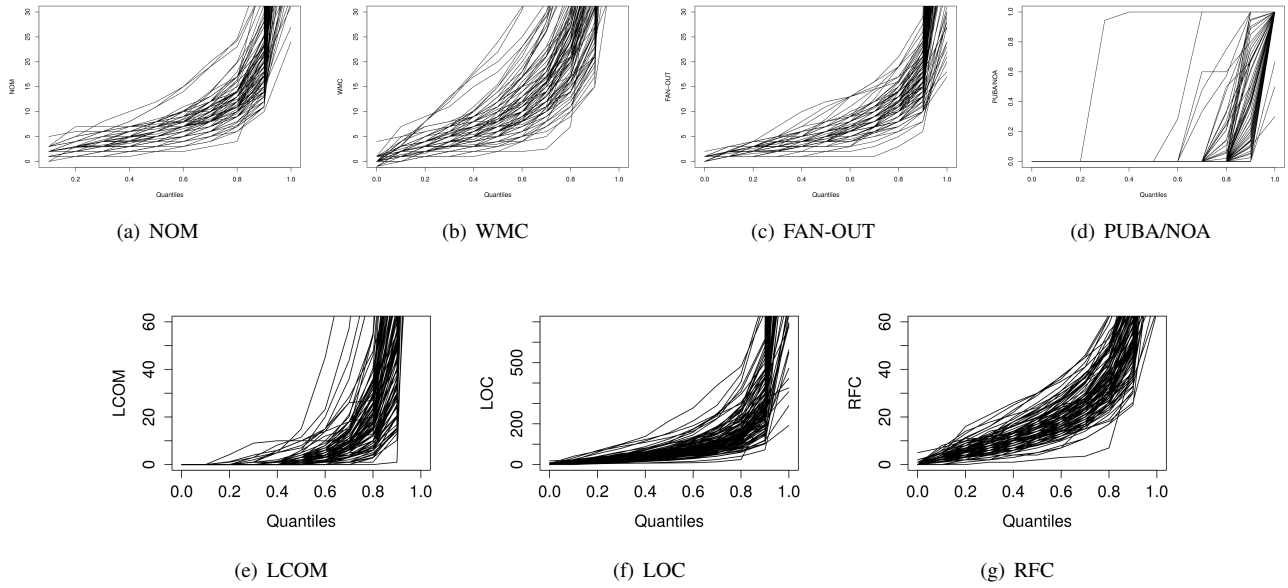
followed by most systems in the considered corpus (in fact, it is followed by 98 out of 106 systems). Second, it is not based on rather lenient upper bounds. In other words, limiting NOA to eight attributes is compatible with an *idealized design rule*. For example, there are thresholds proposed by experts that recommend an upper limit of 10 attributes [7].

To illustrate the classes that do not follow the proposed relative threshold, Table I presents the ten classes with the highest number of attributes in our corpus (considering the 98 systems that follow the proposed threshold and only the biggest class of each system). As we can observe, classes with high NOA values are usually Data Classes [12], used to store global constants, like error messages in the AspectJ compiler or bytecode opcodes in the case of the Jasml disassembler.

TABLE I
Classes with highest NOA Values

System	Class	NOA
GeoTools	gml3.GML	907
JasperReports	engine.xml.JRXmlConstants	600
Xalan	templates.Constants	334
Derby	impl.drda.CodePoint	324
AspectJ	core.util.Messages	317
Jasml	classes.Constants	301
POI	ddf.EscherProperties	275
DrJava	ui.MainFrame	266
RSSOwl	internal.dialogs.Messages	225
MegaMek	ui.swing.RandomMapDialog	216

In the corpus there are eight systems (7.5%) that do not follow the relative threshold. For example, in the JMoney system, 39.3% of the classes have more than 8 attributes. In this system, except for a single class, all other classes with NOA > 8 are related to GUI concerns. For example, the `AccountEntriesPanel` class has 37 attributes, including 25 attributes with types provided by the Swing framework. Another non-compliant system is JTOpen, a middleware for accessing applications running in IBM AS/400 hardware platforms. In this case, we counted 414 classes (25.2%) with NOA > 8, which basically implement the communication protocol with the AS/400 operating system. Therefore, the non-compliant behavior is probably due to the complexity of JTOpen's domain.



(a) NOM (b) WMC (c) FAN-OUT (d) PUBA/NOA

(e) LCOM (f) LOC (g) RFC

Fig. 4. Quantile functions

IV. Extensive Study

In this section, we report an extensive study, through which we apply our method to extract relative thresholds for seven source code metrics. This study also includes a subcorpus analysis (Section IV-D), a historical analysis (Section IV-E), and an inequality analysis (Section IV-F).

A. Metrics

In this study, we used seven metrics related to distinct factors affecting the internal quality of object-oriented systems: Number of methods (NOM), Number of Lines of Code (LOC), FAN-OUT, Response For a Class (RFC), Weighted Method Count (WMC), Lack of Cohesion in Methods (LCOM), and the ratio between Number of Public Attributes and Number of Attributes (PUBA/NOA).

B. Dataset and Study Setup

We used the Qualitas Corpus (version 20101126r), which is a dataset with 106 open-source Java-based systems, specially created for empirical research in software engineering [11]. Besides, we used the Moose platform to compute the values of the metrics for each class of each system [13]. Particularly, we use VerveineJ—a Moose application—to parse the source code of each system and to generate MSE files, which is the format supported by Moose to persist source code models. We also implemented a tool that receives as input CVS files with the metric data generated by Moose and computes the relative thresholds, using the method described in Section II.

Although the literature reports that object-oriented metrics usually follow heavy-tailed distributions [8, 10], we decided to check ourselves whether the metric values we extracted present this behavior. For this purpose, we used the EasyFit tool[1] to reveal the distribution that best describes our values. We configured EasyFit to rely on the Kolmogorov-Smirnov Test to compare our metrics data against reference probability distributions. Following a classification suggested by Foss et. al [14], we considered the metric values extracted for the classes of a given system as heavy-tailed when the "best-fit" distribution returned by EasyFit was Power Law, Weibull, Lognormal, Cauchy, Pareto or Exponential. Table II reports the percentage of systems whose metric values were classified as heavy-tailed. The extracted values followed heavy-tailed distributions in at least 94.1% of the systems in our corpus.

TABLE II
Percentage of systems with heavy-tailed metric values distributions

Metrics	% Heavy-Tailed
NOM	100.0
LOC	96.2
FAN-OUT	99.1
RFC	99.1
WMC	100.0
PUBA/NOA	98.1
LCOM	94.1

Figure 4 shows the quantile functions for the considered metric values. In this figure, the x-axis represents the quantiles and the y-axis represents the upper metric values for the classes in the quantile. The figure visually shows that the considered metric values follow heavy-tailed distributions, with most systems having classes with very high metric values in the last quantiles. We can also observe systems with an outlier behavior, due to the presence of high-metrics values even in intermediary quantiles (e.g., 50th or 60th quantiles).

[1]http://www.mathwave.com/products/easyfit.html

C. Extracted Relative Thresholds

Table III shows the relative thresholds derived by our method, considering the same parameters as the ones described in Section III. For each metric, the table shows the values of p and k that characterize the relative thresholds. The table also shows the number of outliers generated by the thresholds, i.e., the number of systems that do not conform to the thresholds. Table IV shows the name of such systems.

TABLE III
RELATIVE THRESHOLDS

Metrics	p	k	# Outliers
NOM	80	16	5
LOC	75	222	11
FAN-OUT	80	15	9
RFC	80	49	11
WMC	80	32	10
PUBA/NOA	75	0.1	8
LCOM	80	36	12

TABLE IV
OUTLIER SYSTEMS

Metrics	Outliers
NOM	Colt, Compiere, HSQLD, JTOpen, Weka
LOC	Colt, Compiere, Derby, Galleon, HSQLD, Ivatagroupware, JFreeChart, JTOpen, Weka, Xalan, Xerces
FAN-OUT	Freecol, Galleon, JAG, JFreeChart, JGraph, JMoney, JPF, SQuirrelSQL, Weka
RFC	Compiere, Freecol, Galleon, HSQLDB, JAG, JFreeChart, JGraph, JMoney, JPF, SQuirrelSQL, Weka
WMC	AspectJ, Colt, Compiere, Derby, HSQLD, IText, JTOpen, mvnForum, Weka, Xerces
PUBA/NOA	ANTLR, AspectJ, Compiere, FitJava, IText, Java Assembling, JSP Wiki, WebMail
LCOM	ANTLR, AspectJ, Axion, Colt, Compiere, HSQLDB, Informa, IText, JFreeChart, JTOpen, Xerces, Weka

We claim again that the proposed relative thresholds represent a commitment between *real* and *idealized* design rules. They express real design practices because they are widely followed by the systems in our corpus. In fact, the number of systems with an outlier behavior ranged from five systems (NOM) to twelve systems (LCOM). The proposed thresholds seem also to represent idealized design rules, as can be observed by the values suggested to the upper limit k. For example, well-known Java code standards recommend that classes should have no more than 20 methods [7] and our method suggested an upper limit of 16 methods.

However, the balance between *real* and *idealized* design rules was achieved by accepting that the thresholds are valid for a representative number of classes, but not for all classes in a system. In fact, the suggested upper limits apply to a percentage p of classes ranging from 75% (LOC and PUBA/NOA) to 80% (NOM, FAN-OUT, RFC, WMC, and LCOM).

For each metric, we manually analyzed the classes located in the "tail" of the considered distributions. Regarding NOM, we concluded that there are at least three categories of classes with many methods. First, many systems have classes that are automatically generated by parser generators or similar tools. Second, we found that GUI classes—particularly classes that redefine methods inherited from `Swing` base classes—also tend to have high NOM values. Finally, there are systems that follow an architecture based on a kernel, whose classes typically have many methods and tend to be classified as God Classes [12]. As expected, we also found out that classes with many methods, tend to have high-values for FAN-OUT, LOC, WMC, RFC, and LCOM. As for the PUBA/NOA, the derived threshold defines that 75% of the classes should have at most 10% of public attributes. By manually inspecting classes which do not follow this threshold, we concluded that generally the `public` fields in such classes, in fact, represent constants, i.e., they are also `static` and `final`.

D. Relative Thresholds for a Subcorpus

To evaluate the impact of the corpus in our results, we recalculated the relative thresholds for a subset of the Qualitas Corpus. Basically, in this subcorpus there are 26 systems (24.5%) classified as *Tools* in the original corpus (which is the most common domain category in the Qualitas Corpus). Table V presents the relative thresholds and the outliers systems, considering this subcorpus as our benchmark data.

TABLE V
RELATIVE THRESHOLDS FOR A SUBCORPUS

Metrics	p	k	# Outliers	Outliers Systems
NOM	80	17	2	Compiere, Weka
LOC	75	260	2	Compiere, Weka
FAN-OUT	85	20	2	JAG, JMoney
RFC	80	56	2	Compiere, Weka
WMC	80	40	2	Compiere, Weka
PUBA/NOA	80	0.3	3	Compiere, Java Assembling, WebMail
LCOM	80	33	3	Compiere, JFreeChart, Weka

In the subcorpus, with the exception of LCOM, the thresholds rely on relatively high values for k. In the same way, for FAN-OUT and PUBA/NOA, the p parameter also relies on slightly high values. For example, the original threshold regarding all systems in the corpus is as follows:

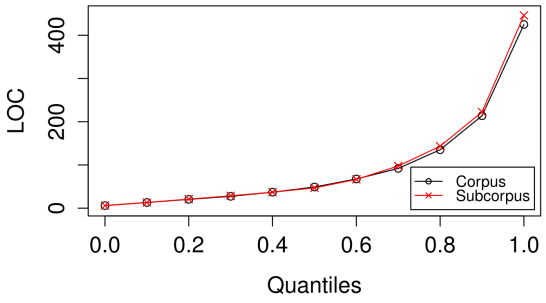
$$\boxed{80\% \text{ of the classes should have FAN-OUT} \leq 15}$$

The same threshold for *Tools* states the following:

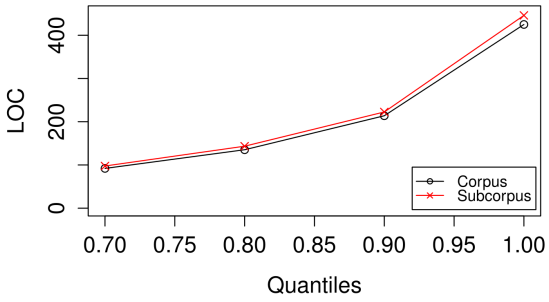
$$\boxed{85\% \text{ of the classes should have FAN-OUT} \leq 20}$$

Figure 5 shows the quantile function regarding the median LOC values for the systems in the Qualitas Corpus and in the *Tools* Subcorpus. Figure 5a shows the function values for all quantiles and Figure 5b shows only the values in the last quantiles. We can observe that the functions are very similar, with a small difference only in the last quantiles. Indeed, in the *Tools* subcorpus, the "tail classes" tend to be bigger than the typical "tail classes" in the whole corpus.

Regarding the outliers, it is worth noticing that the systems classified as outliers (Table V) were also outliers in the whole



(a) All Percentiles



(b) Last Percentiles

Fig. 5. Quantile function for LOC (median values)

corpus (Table IV). On the other hand, some systems initially considered outliers in the whole corpus were not classified as such when we restricted the analysis to the subcorpus. For example, Weka was an outlier for FAN-OUT in the whole corpus, but not in the subcorpus. This observation reinforces the importance of the corpus in methods to extract thresholds empirically from real systems. It also shows that our method was able to reclassify the systems as expected, i.e., when moving from a general to a more homogeneous corpus, some systems were reclassified, but always changing their status from outliers to non-outliers.

E. Historical Analysis

To evaluate whether the proposed thresholds are valid in different versions of the systems under analysis, we performed a historical analysis, considering previous versions of five systems. In this analysis, we considered only the NOM, FAN-OUT, WMC, and PUBA/NOA metrics. Table VI describes the systems and their versions considered in this analysis. Basically, we selected four systems (Lucene, Hibernate, Spring, and PMD) included both in the Qualitas Corpus and in the COMETS Dataset, which is a dataset for empirical studies on software evolution [15]. Essentially, COMETS provides time series for metrics values in intervals of bi-weeks. We extended this dataset to include time series on a new system (Weka), in order to support the analysis also on an outlier system, regarding the NOM, FAN-OUT, and WMC metrics. In Table VI, the period considered in the extraction ends exactly in the bi-week just before the release of the version available in the Qualitas Corpus, i.e., the version we considered to extract the relative thresholds.

TABLE VI
SYSTEMS CONSIDERED IN THE HISTORICAL ANALYSIS

System	Period	Versions
Lucene	01/01/2005—10/04/2008	99
Hibernate	06/13/2007—10/10/2010	82
Spring	12/17/2003—11/25/2009	156
PMD	06/22/2002—14/08/2009	175
Weka	11/16/2008—07/09/2010	45

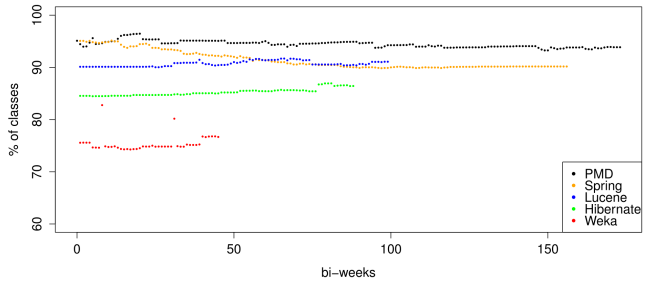
Figure 6 plots the percentage of classes in each version and system considered in this analysis respecting the proposed upper limit (parameter k) in the relative thresholds. We can observe that the proposed thresholds seem to capture an enduring design practice in the considered systems. More specifically, the systems not initially considered as outliers (PMD, Spring, Lucene, and Hibernate) presented the same behavior since the first considered version, in the case of the four metrics. A similar observation holds in the case of Weka. Along the extracted versions, this system did not changed its status, both in the case of the metrics it was classified as an outlier (NOM, FAN-OUT, and WMC), and also for the metric it is not an outlier (PUBA/NOA).

F. Inequality Analysis

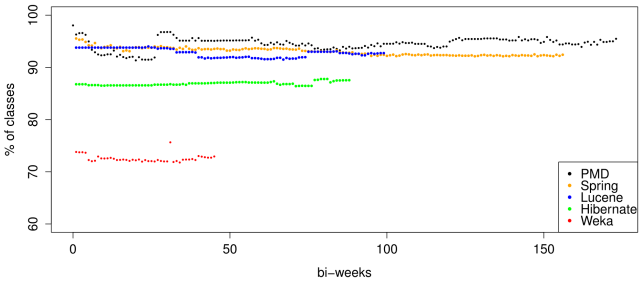
We evaluated the dispersion of the metric values in the systems respecting the proposed thresholds, using the Gini coefficient. Gini is a coefficient widely used by economists to express the inequality of income in a population [16]. The coefficient ranges between 0 (perfect equality, when everyone has exactly the same income) to 1 (perfect inequality, when a single person concentrates all the income). Gini has been applied in the context of software evolution and software metrics [16, 17], although not exactly to evaluate the reduction in inequality achieved by following metric thresholds.

In the analysis we will consider the distributions of NOM values in the original corpus. First, we calculated the Gini coefficient considering the whole population of classes in each system. Next, we recalculated the coefficient for the classes respecting the upper threshold of 16 methods. In both cases, we excluded the systems with an outlier behavior, since our goal is to reveal the degree of inequality in systems respecting our approach. The boxplots in Figure 7 summarizes the Gini results in our systems. As we can observe, the median Gini coefficient considering the whole population of classes in each system is 0.52. By considering only classes with 16 or less methods, the median coefficient is reduced to 0.46. In fact, this reduction in dispersion is expected, since we removed the high values in the long tail.

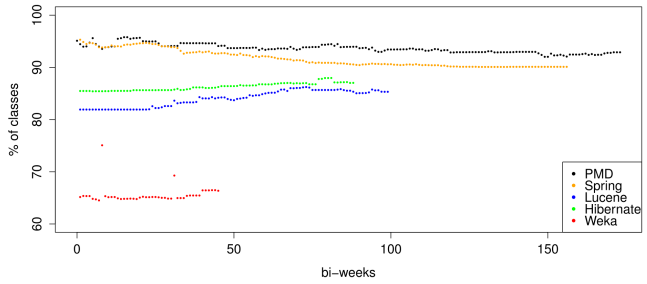
We also analyzed the outliers in the sample filtered by the proposed threshold. As observed in the right boxplot in Figure 7,



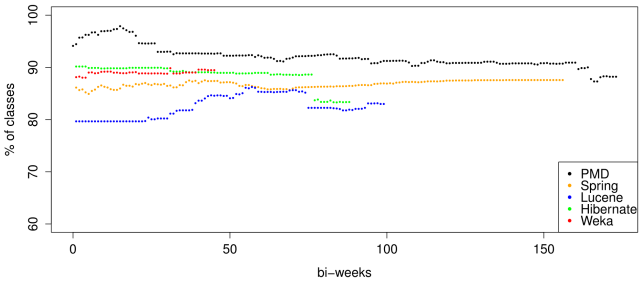
(a) NOM ($p = 80, k = 16$)



(b) FAN-OUT ($p = 80, k = 15$)



(c) WMC ($p = 75, k = 32$)



(d) PUBA/NOA ($p = 75, k = 0.1$)

Fig. 6. Historical Analysis

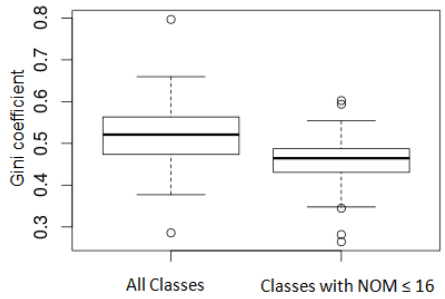


Fig. 7. Inequality Analysis using Gini coefficients

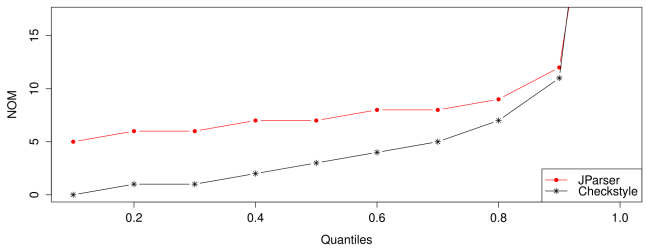


Fig. 8. Quantile functions for outliers regarding the Gini values

we have outliers due to very equal distributions (Gini < 0.35) and also outliers due to very unequal distributions (Gini > 0.55). For example, JParser is an example of the first case (Gini=0.26) and CheckStyle is an example of the second case (Gini=0.60). Figure 8 shows the quantile functions for these two systems. We can see that most classes in JParser respecting the proposed threshold have between 5 to 10 methods, while in CheckStyle we have a representative number of classes with less than five methods, between 5 to 10 methods, and also with more than 10 methods.

Although JParser and CheckStyle have very different Gini coefficients, we can not claim they are outliers in terms of software quality. In other words, a system with classes ranging from 5 to 10 methods (JParser) seems to be not very different than a system having classes with 1 to 16 methods (Checkstyle), at least in terms of their internal software quality.

Therefore, as revealed by the Gini coefficients, the inequality analysis shows that there are different distributions of methods per class among the systems that follow the proposed thresholds. However, such differences do not seem to have major impacts in terms of software quality. More specifically, at least in our corpus, we have not found degenerate distributions, both in terms of equality or inequality, e.g., a system with all classes having exactly a single method or a system with half of the classes having $k - 1$ methods and half of the classes having very few methods. Although such distributions may respect our thresholds, they would certainly reveal serious design problems. On the other hand, it is hard to believe that distributions like that are possible in practice.

G. Threats to Validity

In this extended study, we have not manually analyzed the systems in the Qualitas Corpus to remove for example test classes or classes generated automatically by tools like parsers generators, as usually recommended [18]. However, our central goal with this study was not establishing an industrial software quality benchmark, but to illustrate the use of our method in a real software corpus, including a discussion on its main properties, like sensibility to the systems in the corpus and historical validity of the extracted thresholds. Considering such goals, we consider that the removal of test classes and classes generated automatically is less critical.

V. Discussion

In this section, we discuss our method considering four aspects: (a) adherence to requirements proposed to evaluate metrics aggregation techniques; (b) robustness to staircase effects; (c) tolerance to bad smells; and (d) statistical properties.

A. Requirements

Mordal et al. defined a set of requirements to characterize software metrics aggregation techniques [19]. We reused these categories to discuss our method mainly because metric aggregation and metric thresholds ultimately share the same goal, i.e., to support quality assessment at the level of systems.

In the following discussion, we consider the two most important categories in this characterization (*must* and *should* requirements).

Must Requirements:

- *Aggregation*: Relative thresholds can be used to aggregate low level metric values (typically in the level of classes) and therefore to evaluate the quality of an entire project.
- *Composition*: In our method, metrics should be first composed and then aggregated in the form of a relative threshold. For example, PUBA/NOA—used in the study in Section IV—is an example of a composed metric.

Should Requirements:

- *Highlight problems:* By their very nature, relative thresholds can indicate design problems under accumulation in the classes of object-oriented systems.
- *Do not hide progress:* The motivation behind this requirement is to reveal typical problems when using aggregation by averaging. More specifically, averages may fail due to a tendency to hide outliers. On the other hand, we argue that our method automatically highlights the presence of outliers above an expected value.
- *Decomposability:* Given a partition of the system under evaluation, it is straightforward to select the partitions that concentrate more classes not respecting the proposed thresholds. Possible partition criteria include package hierarchy, programming language, maintainers, etc.
- *Composition before Aggregation:* As explained before, metrics should be composed first to preserve the intended semantic of the composition.
- *Aggregation Range:* This requirement establishes that the aggregation should work in a continuous scale, preferably left and right-bounded. In fact, our relative thresholds can be viewed as predicates that are followed or not by a given system. Therefore, we do not strictly follow this requirement. We discuss the consequence of this fact in Section V-B.
- *Symmetry:* Our final results do not depend on any specific order, i.e., the classes can be evaluated in any order.

B. Staircase Effects

Staircase effects are a common drawback of aggregation techniques based on thresholds [19]. In our context, these effects denote the situation where small refactorings in a class may imply in a change of threshold level, while more important ones do not elevate the class to a new category. To illustrate the scenario, suppose a system with n classes not following a given relative threshold. Suppose also that by refactoring a single class the system will start to follow the threshold. Although the scenarios before and after the refactoring are not very different regarding the global quality of the system, after the refactoring the system's status changes, according to the proposed threshold. Furthermore, when deciding which class to refactor, it is possible that a maintainer just selects the class more closer to the upper parameter of the relative threshold (i.e., the "easiest" class to refactor).

Although subjected to staircase effects, we argue that any evaluation based on metrics—including the ones considering continuous scales—are to some extent subjected to quality treatments. In fact, treating values is a common pitfall when using metrics, which can only be avoided by making developers aware of the goals motivating their adoption [20].

C. Tolerance to Bad Smells

Because the thresholds tolerate a percentage of classes with high metric values, it is possible that they in fact represent bad smells, like God Class, Data Class, etc. [12]. However, when limited to a small number of classes—as required by our relative thresholds—our claim is that bad smells do not constitute a threat to the quality of the entire project nor an indication of an excessive technical debt. Stated otherwise, our goal is to raise quality alerts when bad smells change their status towards a disseminated and recurring design practice.

D. Statistical Properties

In the method to extract relative thresholds, the median of a high percentile (as defined by the $Tail$ parameter) is used to penalize upper limits that do not reflect the accepted semantics for a given metric values. We acknowledge that the use of the median in this case is not strictly recommended, because we never checked whether the $Tail$-th percentiles follow a normal distribution. However, our intention was not to compute an expected value for the statistical distribution, but simply to penalize compliance rates based on lenient upper limits, i.e., limits that are not observed at least in half of the systems in our corpus.

VI. Related Work

In this section, we discuss work related to our method following a division in two groups: (a) thresholds definitions; and (b) statistical analysis.

A. Thresholds Definitions

Alves et al. proposed an empirical method to derive threshold values for source code metrics from a benchmark of systems [4]. Their ultimate goal was to use the extract thresholds to build a maintainability assessment model [21, 22]. In the proposed method, metric values for a given program entity are first weighted according to the size of the entities in terms of lines of code (LOC), in order to generate a new distribution where variations in the metrics values are more clear. After this step, the method relies on quality profiles to rank entities according to four categories: low risk (0 to 70th percentiles), moderate risk (70th to 80th percentiles), high risk (80th to 90th percentiles), and very-high risk (90th percentile). In a more recent work, Alves et al. improved their method to include the calibration of mappings from code-level measurements to system-level ratings, using an N-point rating system [23]. On the other hand, in our method, we do not use weighting by LOC, and we have only two profile categories (respecting or not the relative thresholds). Moreover, our goal is the extraction of relative thresholds that by construction tolerate high-risk classes, assuming they are natural in heavy-tailed distributions. However, high-risk classes should not exceed a percentage of the whole population of classes.

Ferreira et al. defined thresholds for six object-oriented metrics using a benchmark of 40 Java systems [5]. Using the EasyFit tool, they also concluded that the metric values, except for DIT, follow heavy-tailed distributions. After this conclusion, the authors relied on their own experience to establish three threshold ranks: i) good, which refers to most common values; ii) regular, which refers to values with low frequency, but that are not irrelevant; and iii) bad, which refers to values with rare occurrences. However, they do not establish the percentage of classes tolerated in these categories.

Chidamber et al. analysed a set of metrics in order to assess their usefulness for practicing managers [24]. For this, they relied on empirical data relating the metrics to productivity, rework effort, and design effort on three commercial object-oriented systems. In common with our work, the authors propose that threshold values should not be defined a priori, but rather should come from benchmark data. Following Pareto's 80/20 heuristic, they established that a "high" value for a metric can be defined no lower than the 80th percentile. In contrast, our method does not use a defined percentile as threshold and we also generate a single threshold for the entire benchmark.

Herbold et al. used a machine learning algorithm for calculating thresholds [6]. In their approach, classes and methods are classified as respecting or not the computed thresholds. Shatnawi et al. [25] and Catal et al. [26] used Receiver-Operating Characteristic curves (ROC curves) to derive thresholds. Shatnawi et al. derived thresholds to predict the existence of bugs in different error categories using three releases of the Eclipse project. Catal et al. proposed a noise detection algorithm based on software metric thresholds values. On the other hand, Yoon et al. [27] used the K-means Clustering algorithm to derive metric thresholds. However, this algorithm requires an input parameter that affects both the performance and the accuracy of the results. Nevertheless, in all these cases the proposed thresholds are absolute. In contrast, our method derives relative thresholds.

B. Statistical Analysis

Wheeldon et al. analyzed 11 coupling metrics using three Java systems and concluded that their values follow a Power Law distribution [28]. Baxter et al. [8] analyzed 17 metrics in 56 Java systems for verifying their internal structure. The authors reported that most metrics follow power-laws. Louridas et al. analyzed coupling metrics using 11 systems developed in multiple languages (C, Perl, Ruby, and Java) [9]. The authors concluded that most metrics are in conformity with heavy-tailed distributions, independently of programming language. Studies conducted by Potanin et al. [29] and Taube-Schock et al. [10] confirm such results, but for coupling metrics. Concas et al. analyzed 10 metrics using three large Java and Smalltalk systems [30]. Their findings indicate that large OO systems also follow heavy-tailed distributions.

In summary, the aforementioned studies suggest that non-Gaussian distributions are common in the case of source code metric values. Therefore, our method does not assume a distribution that is rarely observed in real-world systems, specifically in the case of source code metrics. On the other hand, such studies do not propose a clear roadmap to apply their findings in practical software quality assessments.

VII. Conclusion

Source code metric values usually follow heavy-tailed distributions [8, 9]. Therefore, it is natural to observe in every system a percentage of classes not respecting a given threshold. In this paper, we proposed the notion of relative thresholds to deal with such metric distributions. Our approach explicitly indicates that thresholds should be valid for most but not for all classes in object-oriented systems. We proposed a method that extracts relative thresholds from benchmark data and we evaluated this method in the Qualitas Corpus. We argued that the extracted thresholds represent an interesting balance between real and idealized design rules.

We envision a scenario where the proposed relative thresholds are used to measure the technical debt in a system [31]. Resembling the notion of financial debt, this metaphor describes the "debt" incurred when developers make quick-and-dirty implementations, targeting a short-term solution. However, in the long term, an accumulated technical debt may cause relevant maintenance costs [32]. On the other hand, it is widely accepted that technical debt can not be completely avoided, although it should always be explicit [33]. In this context, we consider that the notion of relative thresholds can be used to control and monitor the technical debt in a system and to raise an

alarm when the debt reaches a dangerous level, i.e., when the proposed relative thresholds are violated.

Also as future work, we have plans to apply our approach on new software metrics, including non-source code based metrics, such as process metrics. We intend to evaluate our approach with other systems, possibly using the portfolio of a real software development organization. We also intend to extract relative thresholds for different contexts, such as systems of different sizes and systems implemented in different programming languages. Finally, we intend to investigate the impact of not following our thresholds on software quality properties, like maintainability.

ACKNOWLEDGMENTS

Our research is supported by CAPES, FAPEMIG, and CNPq.

REFERENCES

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

[2] F. B. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *4th International Conference of Software Quality (ICSQ)*, 1994, pp. 3–5.

[3] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[4] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *26th IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[5] K. Ferreira, M. Bigonha, R. Bigonha, L. Mendes, and H. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2011.

[6] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Journal of Empirical Software Engineering*, vol. 16, no. 6, pp. 812–841, 2011.

[7] California Institute of Technology, "JPL institutional coding standard for the Java programming language," Tech. Rep., 2010.

[8] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of Java software," in *21th Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006, pp. 397–412.

[9] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, pp. 1–26, 2008.

[10] C. Taube-Schock, R. Walker, and I. Witten, "Can we avoid high coupling?" in *25th European Conference on Object-Oriented Programming (ECOOP)*, 2011, pp. 204–228.

[11] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas corpus: A curated collection of Java code for empirical studies," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2010, pp. 336–345.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[13] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," *Software Engineering Notes*, vol. 30, no. 5, pp. 1–10, 2005.

[14] S. Foss, D. Korshunov, and S. Zachary, *An Introduction to Heavy-Tailed and Subexponential Distributions*. Springer-Verlag, 2011.

[15] C. Couto, C. Maffort, R. Garcia, and M. T. Valente, "COMETS: a dataset for empirical research on software evolution using source code metrics and time series analysis," *Software Engineering Notes*, vol. 38, no. 1, pp. 1–3, 2013.

[16] B. Vasilescu, A. Serebrenik, and M. van den Brand, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in *27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 313–322.

[17] R. Vasa, M. Lumpe, P. Branchand, and O. Nierstrasz, "Comparative analysis of evolving software systems using the Gini coefficient," in *25th IEEE International Conference on Software Maintenance (ICSM)*, 2009, pp. 179–188.

[18] T. L. Alves, "Categories of source code in industrial systems," in *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, pp. 335–338.

[19] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Practical software quality metrics aggregation," *Software Maintenance and Evolution: Research and Practice*, pp. 1–19, 2013.

[20] E. Bouwers, J. Visser, and A. van Deursen, "Getting what you measure," *Communications of the ACM*, vol. 55, no. 7, pp. 54–59, 2012.

[21] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[22] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *6th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2007, pp. 30–39.

[23] T. L. Alves, J. P. Correia, and J. Visser, "Benchmark-based aggregation of metrics to ratings," in *21th International Workshop on Software Measurement (IWSM)*, 2011, pp. 20–29.

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

[25] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 1, pp. 1–16, 2010.

[26] C. Catal, O. Alan, and K. Balkan, "Class noise detection based on software metrics and ROC curves," *Information Sciences*, vol. 181, no. 21, pp. 4867 – 4877, 2011.

[27] K.-A. Yoon, O.-S. Kwon, and D.-H. Bae, "An approach to outlier detection of software measurement data using the k-means clustering method," in *1st Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2007, pp. 443–445.

[28] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," in *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003, pp. 1–10.

[29] A. Potanin, J. Noble, M. Frean, and R. Biddle, "Scale-free geometry in OO programs," *Communications of the ACM*, vol. 48, pp. 99–103, 2005.

[30] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-Laws in a Large Object-Oriented Software System," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 687–708, 2007.

[31] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[32] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, p. 9, 2012.

[33] N. Zazworka, A. Vetro?, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Journal*, pp. 1–24, 2013.