Validating Metric Thresholds with Developers: An Early Result

Paloma Oliveira*†, Marco Tulio Valente*, Alexandre Bergel‡ and Alexander Serebrenik§
*UFMG, Belo Horizonte, Brazil
†IFMG, Formiga, Brazil
‡Pleaid Lab, Department of Computer Science (DCC), University of Chile, Santiago, Chile
§Eindhoven University of Technology, Eindhoven, The Netherlands
paloma.oliveira@ifmg.edu.br, mtov@dcc.ufmg.br, abergel@dcc.uchile.cl, a.serebrenik@tue.nl

***Abstract*—Thresholds are essential for promoting source code metrics as an effective instrument to control the internal quality of software applications. However, little is known about the relation between software quality as identified by metric thresholds and as perceived by real developers. In this paper, we report the first results of a study designed to validate a technique that extracts relative metric thresholds from benchmark data. We use this technique to extract thresholds from a benchmark of 79 Pharo/Smalltalk applications, which are validated with five experts and 25 developers. Our preliminary results indicate that good quality applications—as cited by experts—respect metric thresholds. In contrast, we observed that noncompliant applications are not largely viewed as requiring more effort to maintain than other applications.**

***Index Terms*—Source Code Metrics; Relative Thresholds; Software Quality; Software Measurement; Empirical Studies.**

I. Introduction

Source code metrics are common in evaluating quality and specifically maintainability of software applications [1]. Nevertheless, metric values require careful interpretation, *e.g.*, when checking good/bad software design or presence/absence of maintainability problems. Such an interpretation is often based on thresholds [2], [3]: methods, classes, components with metrics values that do not exceed a given threshold are deemed good or maintainable while those with metrics values exceeding the threshold are deemed bad or suffering from maintainability problems. However, software metrics are known to be right-skewed [2], [4]. Hence, any predefined metric threshold can be expect to be violated by *some* methods, classes or components in large enough applications [5].

To address the challenge of identifying thresholds that have to be followed by many but not all methods, classes or components we have introduced in previous work the notion of a *relative threshold* [6], [7]. *Relative thresholds* are represented by pairs (p, k) such that $p\%$ of the classes in an application should have $M \leq k$, where M is a given source code metric and p is the minimal percentage of classes in each application that should respect the upper limit k.

However, we did not validate to what extent the notion of relative thresholds is able to effectively distinguish applications with maintainability problems. To tackle this question, this paper reports early results of a study designed to validate metric thresholds with developers. Specifically, we extract relative thresholds for 79 Pharo applications. Pharo is a Smalltalk-like language, with an active community of developers and researchers. We leverage our ability to directly access expert developers on Pharo to check whether applications with good (or bad) maintainability respect (or no) the thresholds we initially extracted in the study. Basically, we investigate three research questions:

RQ #1 Do applications perceived as **well-written** by expert developers respect the derived relative thresholds?

RQ #2 Do applications perceived as **poorly-written** by expert developers do not respect the derived relative thresholds?

RQ #3 Do **noncompliant** applications require more effort to maintain? By noncompliant we refer to applications that do not respect the relative thresholds on multiple metrics.

Our contributions are twofold. First, we report the extraction of relative thresholds for a *Corpus* composed of 79 applications implemented in Pharo. Second, we describe a preliminary validation study with expert developers, who are the right experts to check whether metric thresholds are indeed able to infer maintainability and design problems.

The remainder of this paper is organized as follows. Section II summarizes the proposed technique to extract relative thresholds. Section III describes the design of our empirical study. Next, Section IV reports our preliminary findings. Section V presents threats to validity and Sections VI and VII discuss related and further work, respectively.

II. Relative Thresholds

In this section we summarize the proposed technique for extracting relative thresholds from a *Corpus*, a benchmark of software applications [6], [7]. This technique derives thresholds represented by pairs $[p, k]$, such that $p\%$ of the classes in an application should have $M \leq k$. Therefore, a relative threshold tolerates $(100 - p)\%$ of classes with $M > k$. As an example, we can derive a relative threshold as follows: at least 75% of the classes in an application should have NOM $\leq$ 29.

A relative threshold $[p, k]$ is derived using two functions, called *ComplianceRate* and *ComplianceRatePenalty*. The function *ComplianceRate*$[p, k]$ returns the percentage of applications in the *Corpus* that follow the relative threshold defined by the pair $[p, k]$. However, this function on its own is not

sufficient to optimize p and k. Hence, we introduce the notion of *penalties* to find the values of p and k. We penalize a *ComplianceRate* function in two situations. The first penalty fosters the selection of thresholds followed by at least 90% of the applications in the *Corpus*. The goal is to derive thresholds that reflect real design rules, which are widely common in the *Corpus*. Furthermore, *ComplianceRate*$[p, k]$ receives a second penalty whenever k is greater than metric values that are perceived as being very high. Finally, the *ComplianceRatePenalty* function is the sum of *penalty*$_1[p, k]$ and *penalty*$_2[k]$. A derived relative threshold is the one with the lowest *ComplianceRatePenalty*$[p, k]$. A detailed description of our technique is out of the scope of this paper and is available elsewhere [6], [7].

III. Study Design

In this section we present the *Corpus* and the considered source code metrics (Section III-A). Next, we describe the methodology and participants of our study (Section III-B).

A. Corpus and Metrics

In order to validate our notion of relative thresholds with software developers, we use a *Corpus* of 79 Pharo applications[1]. We initially select 39 applications found in the Pharo standard distribution. From these 39 applications, 18 may be considered as legacy, although they are intensively in use, covered by unit tests, and have received numerous contributions by the community. In addition, we select 40 applications from the Pharo forge[2]. These additional applications are selected based on their size, popularity, activity, and relevance for the Pharo ecosystem. Most of these applications are part of specialized distributions of Pharo (namely *Moose* and *Seaside*), confirming their relevance and maturity. For this study, tests classes were removed because they usually have a structure radically different from production code.

In this study, we make a first attempt to validate relative metric hresholds for the following four source code metrics computed by the Moose software analysis platform[3]: (a) Number of Attributes (NOA)—Moose computes this metric by counting all attributes in the class; (b) Number of Methods (NOM)—Moose computes this metric by counting all methods in the class, including constructors, getters, and setters; (c) Number of Provider Classes (FAN-OUT)—Moose computes this metric by considering all types of class dependencies (method calls, etc.); and (d) Weighted Method Count (WMC)—Moose computes this metric as the sum of the cyclomatic complexity of each method in a class.

B. Methodology and Participants

Recall that our goal is to validate the notion of relative thresholds with Pharo practitioners. To achieve this goal, we initially conducted a survey involving five Pharo experts (*i.e.*, people deeply committed to the Pharo development and success) and 25 Pharo maintainers (*i.e.*, people in charge of incorporating application improvements and producing new releases).

Specifically, to answer RQ #1 and RQ #2, we asked five experts in Pharo, which are members of the Pharo board, to provide examples of applications that are "well-written" and "not well-written". The choice of these terms is based on the outcome of a pilot study, which indicated that "well-written" and "not well-written" are largely understood by practitioners as opposed to terms such as "maintainability", that practitioners had difficulty on interpreting.

To answer RQ #3, we focus on applications that do not respect the derived relative thresholds for at least two metrics[4]. We call such applications *noncompliant* and we interviewed the top maintainers of each one. A top maintainer is a developer that has written most of the methods found in the last release of the application. Specifically, we identified the five top maintainers in each noncompliant application by ranking authors of each application according to the number of contributed methods (*i.e.*, defined or modified methods). We asked these maintainers the following question: *Compared to other applications you work with, the application in question requires (a) more effort to maintain; (b) comparable effort to maintain; (c) less effort to maintain.*

IV. Findings

In this section, we first present the relative thresholds for the source code metrics considered in this paper, derived using the Pharo Corpus (Section IV-A). Next, we describe the results of our research questions (Sections IV-B to IV-D).

A. Relative Thresholds for the Pharo Corpus

Table I presents the relative thresholds derived by our technique. For each metric, the table shows the values of p and k that define a relative threshold. It also shows the number of applications that do not respect these thresholds. We can observe that the upper limit k of the derived relative thresholds are valid for a large number of classes (parameter p), but not for all classes in an application. In fact, the value of p ranges from 75% to 80%. The number of applications that do not respect the relative thresholds range from six (FAN-OUT) to 14 (WMC), *i.e.*, from 7.6% to 17.7% of the applications in the *Corpus*.

TABLE I: Relative Thresholds for Pharo

Metrics	p	k	# of applications that do not respect the relative thresholds
NOA	75	5	9
NOM	75	29	11
FAN-OUT	80	9	6
WMC	75	46	14

[1]A detailed description is available at http://aserg.labsoft.dcc.ufmg.br/pharo-dataset

[2]http://smalltalkhub.com, verified on 06/15/2015.

[3]http://www.moosetechnology.org/, verified on 06/15/2015

[4]We did not consider a single metric to avoid the "One-track metric" anti-pattern, which occurs when a single metric is used to measure software quality [8].

Table II presents nine applications considered as noncompliant, *i.e.*, applications that do not respect the relative thresholds for two or more metrics, as described in Section III-B.

TABLE II: Noncompliant applications

Noncompliant	Metrics			
	NOA	NOM	FAN-OUT	WMC
Collections		✓		✓
ComandShell		✓	✓	✓
Files			✓	✓
Graphics	✓	✓	✓	✓
Kernel		✓		✓
Manifest	✓	✓		✓
Morphic		✓		✓
Shout	✓	✓	✓	✓
Tools	✓	✓	✓	✓

*B. RQ 1: Do applications perceived as **well-written** by the expert developers respect the derived relative thresholds?*

To answer this question, we asked five Pharo experts to indicate well-written Pharo applications. Table III presents these applications, including a brief description, and the experts that elected the application. Among the seven applications named by the experts, *Roassal* and *Zinc* belong to the *Corpus*. We claim that this overlap does not affect validity of our analysis. In fact, benchmark-based techniques to derive thresholds depend on a balanced corpus, including both well and poorly-written applications. In other words, the overlapping shows that our *Corpus* includes well-written applications, but as expected the Pharo ecosystem also has other well-written applications.

TABLE III: Well-written applications

Systems	Description	Voted by	Corpus
PetitParser	Parser framework	Expert #1	
PharoLauncher	Platform to manage Pharo images	Expert #2	
Pillar	Markup language and tools	Expert #2	
Roassal	Visualization engine	Expert #3	✓
Seaside	Web framework	Expert #4	
SystemLogger	Log framework	Expert #5	
Zinc	HTTP framework	Expert #5	✓

For each application, we evaluate their percentage of classes respecting the k-value of the proposed relative threshold. The results are summarized in Table IV. For instance, the relative threshold for NOA is $(75, 5)$ and the table shows that 100% of the classes of *PetitParser* have five attributes or less.

As can be observed in Table IV, the well-written applications respect the proposed relative thresholds for all metrics with the notable exception of FAN-OUT. The only applications that respect the relative threshold for FAN-OUT are *SystemLogger* and *Zinc*. To explain this fact, we investigated the distribution of the FAN-OUT values in the *Corpus* and in the well-written applications reported by the Pharo experts. Figure 1 shows the quantile functions for the FAN-OUT values, *i.e.*, the x-axis represents the quantiles and the y-axis represents the upper metric values for the classes in the quantile. The applications that do not respect the relative thresholds for FAN-OUT, *i.e.*, *PetitParser*, *PharoLauncher*, *Pillar*, *Roassal*, and *Seaside*, are represented by dashed lines, while the remaining applications (*Corpus* and well-written applications that respect the thresholds) are represented by solid lines. We can observe that the applications that do not respect the proposed thresholds have very different distribution of FAN-OUT values than applications in our *Corpus*.

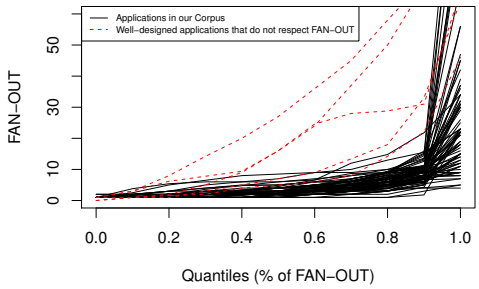


Fig. 1: FAN-OUT quantiles—dashed lines represent *PetitParser*, *PharoLauncher*, *Pillar*, *Roassal*, and *Seaside*, which are applications perceived as well-written but that do not respect the relative threshold for FAN-OUT

Furthermore, we took a closer look at the way Moose computes FAN-OUT. When determining this metric, Moose considers all types of class dependencies introduced, *e.g.*, by means of inheritance, method calls, static accesses, etc. Therefore, one way for a class to have a high FAN-OUT is to be a client of an extensive inheritance hierarchy with many instances of overridden methods. A preliminary inspection in the source code shows that this is exactly the case of *PetitParser*, *PharoLauncher*, *Pillar*, *Roassal*, and *Seaside*.

Summary of findings: We observe that applications perceived as well-written by the interviewed experts follow the proposed relative thresholds for NOA, NOM, and WMC. However, this does not happen for FAN-OUT, as *SystemLogger* and *Zinc* are the only applications that respect the relative threshold for this metric. The reason for "too many high FAN-OUT" values being present in five well-written applications seems to be the presence of extensive inheritance hierarchies with many instances of overridden methods. This finding stresses the importance of considering multiple metrics when determining whether an application might be problematic from the point of view of its internal quality.

*C. RQ 2: Do applications perceived as **poorly-written** by the expert developers do not respect the derived relative thresholds?*

To answer this second research question, we asked the five experts to indicate poorly-written applications. This question turned out to be much more difficult, since only two experts answered it. Table V presents these applications, including a brief description, and the experts that suggested the application. This difficulty to identify poorly-written applications might be explained by the respondents being rather optimistic than pessimistic, or by them not being comfortable with admitting that some applications are not well-written.

TABLE IV: Percentage of classes in the well-written applications that respect the upper limit k of a relative threshold (underlined values show the cases when the thresholds are not respected).

Metric	p	k	% of classes in the well-written applications with M $\leq$ k						
			PetitParser	PharoLauncher	Pillar	Roassal	Seaside	SystemLogger	Zinc
NOA	75%	5	100%	97%	97%	91%	97%	100%	91%
NOM	75%	29	97%	97%	94%	90%	96%	92%	82%
FAN-OUT	80%	9	41%	74%	62%	24%	41%	100%	81%
WMC	75%	46	97%	99%	95%	93%	96%	92%	82%

TABLE V: Poorly-written applications

Systems	Description	Voted by	Corpus
Metacello	Versioning system	Expert #4	✓
Morphic	Graphical interface framework	Experts #2 and #4	✓

For *Metacello* and *Morphic*, Table VI shows the percentage of classes respecting the k-value of the proposed thresholds.

TABLE VI: Percentage of classes in the poorly-written applications that respect the upper limit k of a threshold (underlined values show the cases when the thresholds are not respected).

Metrics	p	k	% of classes with M $\leq$ k	
			Metacello	*Morphic*
NOA	75%	5	93%	77%
NOM	75%	29	82%	74%
FAN-OUT	80%	9	86%	83%
WMC	75%	46	79%	71%

On the one hand, we found that *Morphic* is a noncompliant application, *i.e.*, it does not respect the proposed relative thresholds for two metrics: NOM and WMC. For example, Expert #2 made the following comments about Morphic:

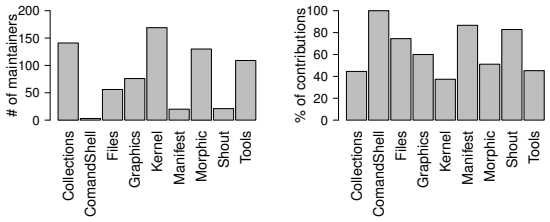
"Morphic is an old system and there is no test and sparse documentation".

On the other hand, *Metacello* respects the relative thresholds for all metrics. This system supports a complex domain-specific language to express intricate relations between different versions of Pharo packages (*e.g.*, this language allows a developer to define that package X depends on version v_1 and v_2 of package Y, only on a platform P). It also takes care of determining cyclic dependencies and identifying proper versions required in presence of multiple dependencies. One Pharo expert argued that the complexity of the versioning domain makes *Metacello* very hard to understand, and there is an on-going effort to replace the system. Therefore, we claim that the perception of *Metacello* as poorly written is more likely to be caused by the inherent complexity of the versioning domain rather than by a problematic design.

Summary of findings: Violation of two relative thresholds in *Morphic* agrees with its design being perceived as problematic. However, this is not the case of *Metacello*. Probably, *Metacello* was cited as poorly-written due to the complexity of its domain.

*D. RQ 3: Do the **noncompliant** applications require more effort to maintain?*

Before answering this third RQ, we analyze the importance of the Top-5 maintainers in the noncompliant applications. Figure 2(a) presents the number of maintainers of each noncompliant. We can observe that this number ranges from three (*ComandShell*) to 169 (*Kernel*). Six out of nine noncompliant applications have more than 50 maintainers, which reinforces the relevance of these applications in the Pharo Ecosystem. Figure 2(b) shows the percentage of contributions by the considered top maintainers. Recall that we ranked the maintainers according to the percentage of their contributions in each noncompliant application. We can observe that the top-5 maintainers contributions range from 37% (*Kernel*) to 100% (*ComandShell*). In five out of nine applications the contributions of these maintainers exceed 60% of the total of the contributions.



(a) Number of maintainers (b) Percentage of contributions by the top-5 maintainers

Fig. 2: Top-5 maintainers analysis in noncompliant applications

To answer RQ #3, we sent out a survey to the 25 maintainers represented in Figure 2(b) and we obtained 11 answers (44%). Based on these answers we calculate the following score expressing the Effort to Maintain (EM) an application S:

$$\boxed{EM = M - L}$$

where M is the number of maintainers that answered that S is more difficult to maintain and L is the number of maintainers that answered that it requires less effort to maintain. Figure 3 shows the EM values for the nine noncompliant applications. Four applications require more effort to maintain ($EM > 0$). To illustrate this fact, we reproduce comments made by a Graphics developer:

"Graphics is a sum of patches over patches without a clear direction on design, with tons of duplicates and several design errors/conflicts. So is a pain to introduce any change there."

Three applications have a maintenance effort that is

comparable to other applications our respondents work with ($EM = 0$). We also observe that $EM < 0$ for two applications. We hypothesize two reasons for these noncompliant applications require less maintenance effort: (a) the metrics used to classify an application as noncompliant do not cover the whole spectrum of properties and requirements the maintainers considered when ranking applications in terms of internal quality; (b) maintainers are usually more wary when judging an application as presenting low quality, as we have learned when investigating the second research question.

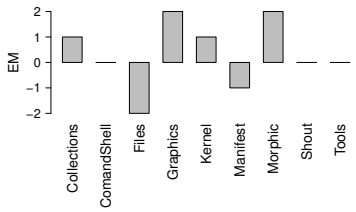


Fig. 3: Effort to Maintain (EM)

Summary of findings: We found that four out of nine noncompliant applications are harder to maintain. Therefore, noncompliant applications are not largely viewed as requiring more effort to maintain than other applications.

V. Threats to Validity

In this section, we discuss possible threats to validity. First, our study participants might not be representative of the whole population of Pharo developers and, in more general terms, of general software developers. Anyway, we interviewed expert developers, with large experience, and who are responsible for the central architectural decisions in their applications. Second, our *Corpus* and metric selections may not be representative enough to evaluate the quality of Pharo applications. However, we at least strive to include well-known and large Pharo applications in the *Corpus*. Moreover, the metrics used in the paper cover important dimensions of an application implementation (size, coupling, and complexity).

VI. Related Work

In this section, we briefly work which explore how developers rate different software quality attributes like readability [9], complexity [10], and coupling [11]. Buse and Weimer explored the concept of code readability and investigate its relation to software quality [9]. They found that readability metrics strongly correlate with measures of software quality. Katzmarski and Koschke investigated whether metrics agree with complexity as perceived by developers [10]. The authors concluded that data-flow metrics seem to better conform to developers' opinions than control-flow metrics. Bavota et al. investigated how coupling metrics align with developers' perception of coupling [11]. They concluded that coupling is not a trivial quality attribute. However, to the best of our knowledge, the study presented in this paper is the first validate metric thresholds technique with developers.

In this paper, we validated a technique for deriving relative thresholds. However, there are other techniques for extracting metric thresholds, using for example, machine learning [12] and benchmarks [2], [3].

VII. Conclusions and Future Work

This paper reported the first results of an empirical study aimed at validating relative metric thresholds with developers. The study was conducted on 79 Pharo applications and using four source code metrics. The results indicate that well-designed applications mentioned by expert respect the relative thresholds. In contrast, we observed that developers usually have difficulties to indicate poorly-designed applications. We also found that four out of nine noncompliant applications are harder to maintain. Therefore, noncompliant applications are not largely viewed as requiring more effort to maintain than other applications. We plan to extend our work as follows:

- By conducting in-depth interviews with at least some of the Pharo experts and maintainers considered in the study. These interviews will help to strength our findings (*e.g.*, to confirm that frameworks are usually noncompliant applications in terms of FAN-OUT), and also to clarify why some noncompliant applications are not perceived as being more difficult to maintain.
- We intend to run a second survey with the maintainers of the noncompliant applications, asking them to provide a direct rating about the maintainability of these applications, without any comparison with other applications they usually work with.
- By considering data from other sources, like mailing lists and bug tracking systems. These sources can help us to better asses the quality of the applications.

ACKNOWLEDGMENTS. Our research is supported by CAPES, FAPEMIG, CNPq and STICAmSud Project 14STIC-02.

References

[1] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
[2] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *ICSM*, 2010, pp. 1–10.
[3] K. Ferreira, M. Bigonha, R. Bigonha, L. Mendes, and H. Almeida, "Identifying thresholds for object-oriented software metrics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, 2011.
[4] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of Java software," in *OOPSLA*, 2006, pp. 397–412.
[5] C. Taube-Schock, R. Walker, and I. Witten, "Can we avoid high coupling?" in *ECOOP*, 2011, pp. 204–228.
[6] P. Oliveira, M. T. Valente, and F. Lima, "Extracting relative thresholds for source code metrics," in *CSMR-WCRE*, 2014, pp. 254–263.
[7] P. Oliveira, F. Lima, M. T. Valente, and A. Serebrenik, "RTTool: Extracting relative thresholds for source code metrics," in *ICSME-Tool Track*, 2014, pp. 629–632.
[8] E. Bouwers, J. Visser, and A. van Deursen, "Getting what you measure," *Commun. ACM*, vol. 55, no. 7, pp. 54–59, 2012.
[9] R. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 546–558, 2010.
[10] B. Katzmarski and R. Koschke, "Program complexity metrics and programmer opinions," in *ICPC*, 2012, pp. 17–26.
[11] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *ICSE*, 2013, pp. 692–701.
[12] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *J. Emp. Softw. Eng.*, vol. 16, no. 6, pp. 812–841, 2011.