DIFFERENT TECHNIQUE OF EXTRACTING THRESHOLD VALUE FOR SOURCE CODE METRICS

Shaitan Singh Meena[1], **Dr. Satwinder Singh**[2],
Centre for computer science and technology, Central university of Punjab, Bathinda, India
Email - shaitansinghmeena@gmail.com[1], satwindercse@gmail.com[2]

Abstract: *Meaningful thresholds are needed for promoting software metrics as an effective instrument to measure the internal quality of systems. The propose the concept of different technique of desgin Thresholds for evaluating metrics data. The disscrebde tehniqiue of proposed thresholds assume that metric thresholds should be followed by most entities. Describe an empirical method for deriving Threshold from a corpus of systems. I also perform an extensive analysis of Threshold technique (i) extracting threshold using traditional techniques; (ii) extracting threshold using error models. (iii) Extracting Thresholds using Clustering Algorithms. with thresholds extracted according to a method used by the software industry.*

Key Words: *Source code metrics, Thresholds, error models, clustering algoritthms.*

1. INTRODUCTION:

The primary goal of software engineering is to produce high quality software. Software quality can be reached using two important concepts: Software Process Quality and Software Product Quality [19]. A software process is a set of activities, practices, methods, and transformations used to develop and to maintain software and the associated products (e.g., project plans, design documents, code, test cases, and user manuals) [19]. The adopted development process reflects in productivity, cost, and in the software quality. Software product quality has been given less importance when compared to other areas of software quality, with exception for testing. For a long time, reliability (as measured in a number of failures) has been the single criteria for gauging software product quality. Software products are getting larger in size and in number of components, where different components exchange information using several interfaces to other components. This means that the overall complexity of the systems grows. It is estimated that 50-80% of the costs of the software project goes to maintenance [11]. This is the reason why it is important for a software company to understand the quality of their products in order to increase efficiency of the software development. One of the challenges of software quality research is to identify how to use metrics to drive the development processes and to improve the software product. Source code metrics can be used to identify possible problems. or chances for improvements in software quality [19]. A variety of metrics to measure source code properties like size, complexity, cohesion, and coupling have been proposed [1, 6, 12, 14]. However, source code metrics are rarely used to support decision making because they are ultimately just numbers that are not easy to interpret [19]. Usually, metrics are classified into three categories: process, project, and product, as described next [11].

- Process metrics: enable the organization to evaluate the development process. They can be used to improve software development and maintenance practices. As examples of process metrics, we can mention function point, change metrics, number of files involved in bug fixing, etc.
- Project or resources metrics: enable the organization to evaluate the progress of a software project. Basically, they describe the project characteristics and execution. As examples of project or resources metrics, we can mention a number of developers, cost, schedule, and productivity.
- Product metrics: enable software engineers to evaluate internal properties of a software product. As examples of product metrics, we can mention size, complexity, coupling, and cohesion.

Chidamber and Kemerer have proposed one of the most widely referenced sets of object- oriented software metrics [5, 6]. Often referred to as the CK metrics suite, it includes six class-based design metrics:

Weighted Methods per Class (WMC): represents the complexity of the class as measured by its methods. The calculation of the metric is given by the sum of the complexity of the methods in the class. According to Chidamber and Kemerer, WMC is an indicator of how much time and effort are required to develop and maintain a given class. Currently, some authors define WMC as the number of methods in the class.

Depth of Inheritance Tree (DIT): indicates the depth of a class in the inheritance tree, which is given by the length of the path from the class to the root of the tree. DIT is nowadays considered an indicator of design complexity. Number of Children (NOC): denotes the number of immediate subclasses of a class. This metric is an indicator of the importance that a class has in the system. If a class has a large number of children, it might, for example, require more tests.

Coupling between Object Class (CBO): indicates the number of classes to which a certain class is coupled to. For Chidamber and Kemerer, a coupling between two classes exists when the methods implemented in one class use methods or instance variables defined by other classes. This metric can be used to reveal design problems. For example, it is widely accepted that excessive coupling is harmful to modular design, because the more independent a class is, easiest is to reuse it in other systems.

Response for a Class (RFC): indicates the number of methods that can be called in response to a message received from a class, defined as the number of methods of the class plus the number of methods invoked by them. RFC is considered an indicator of coupling.

Lack of Cohesion in Methods (LCOM): indicates the lack of cohesion between the methods in a class. Chidamber and Kemerer propose that cohesion between methods can be captured by the use of common instance variables. In this way, LCOM is usually computed as the number of method pairs that have no instance variables in common minus the number of method pairs with common instance variables. Therefore, the smaller the value of LCOM, the more cohesive is the class.

CK metrics cover different internal properties of software systems, such as complexity (WMC), coupling (CBO and RFC), inheritance (DIT and NOC), and cohesion (LCOM). It is also important to state that, there are other object-oriented metrics cited in the literature [1, 14]. Among such metrics, we can mention the number of lines of code (SLOC), number of methods (NOM), number of attributes (NOA), a number of other classes referenced by a class (FAN-OUT), etc. Software metrics have been proposed to analyze and evaluate software by quantitatively capturing a specific characteristic or a view of a software system.

2. LITERATURE REVIEW:

Different methods to derive thresholds. These methods are organized groups: (a) extracting threshold using traditional techniques (b) extracting threshold using error models. (c) Extracting Thresholds using Clustering Algorithms.

(a) **Extracting Thresholds using Traditional Techniques:** Erni and Lewerentz proposed the use of mean (μ) and standard deviation (o) to derive a threshold T from project data [7]. For this, the authors used coupling, complexity, and cohesion metrics. A threshold T is calculated as Tlow = μ+o or Thigh = μ—o, indicating that high or low values of a metric can cause problems, respectively. This method is a common statistical technique which data are normally distributed. However, the authors did not analyze the underlying distribution and only applied it to one system, using three releases. The problem with the use of these methods is that they assume that metric data are assumed to be normally distributed, thus compromising their validity in general. Software metrics generally follow heavy-tailed distributions. Consequently, the use of means and the standard deviation is not adequate.

(b) **extracting threshold using error models**: Shatnawi et al. investigated the use of the ROC curves to extract thresholds for predicting the existence of bugs in different error categories [21]. They performed an experiment using 12 source code metrics and applied the method to three releases of Eclipse. The metrics analyzed were: number of attributes (NOA), number of operations (NOO), lack of cohesion of methods (LCOM), weighted methods complexity (WMC), coupling between objects (CBO), coupling through data abstraction (CTA), coupling through message passing (CTM), response for class (RFC), depth of inheritance hierarchy (DIT), number of child classes (NOC), number of added methods (NOAM), and number of overridden methods (NOOM). Catal et al. [4] developed a noise detection approach that uses threshold values for software metrics in order to capture these noisy instances. The thresholds of Catal et al. were calculated using an adaptation of the Shatnawi et al. [21] threshold calculation technique. They validated the proposed noise detection technique on five public NASA datasets. The results showed that this method is effective for detecting noisy instances. Although Shatnawi et al. and Catal et al. extracted thresholds using ROC curves, this method resulted in three drawbacks in their results. First, threshold values can not be found. Second, for different releases of a system, different thresholds were derived. Third, the methodology does not succeed in deriving monotonic thresholds, i.e., lower thresholds were derived for higher error categories than for lower ones. Benlarbi et al. analyzed the relation of source code metric thresholds and software failures using linear regressions. This study was performed using five CK metrics (WMC, DIT, NOC, CBO, and RFC) and two C++ systems. The authors compared two error probability models, one with a threshold and another without. For the model with a threshold, zero probability of error exists for metric values below the threshold. They concluded that there was no empirical indication supporting the model with a threshold as there was no significant difference among the models. However, this result is only valid for this specific error prediction model and for the metrics the authors took into account. Other models can, potentially, give different results. Duplicated code, overly complex methods, non-cohesive classes, and long parameter lists are possible signs of degradation in the design of software system [15]. These signs are usually known as design flaws, bad smells [10].

(c) **Extracting Thresholds using Clustering Algorithms:** Yoon et al. investigated the use of the K-means clustering algorithm to identify outliers in the data measurements [17]. Outliers can be identified by observations that appear either in isolated clusters (external outliers) or by observations that appear far away from other observations within the same cluster (internal outliers). Oliveira et al. proposed a quantitative approach based on source code metrics to determine similarity in object-oriented systems [13, 16]. This approach also used K-means clustering algorithm to derive thresholds. The thresholds generated by this approach represents profiles of classes of a system. The authors performed two case studies using a dataset with more than 100 Java systems and 23 metrics. However, K-means suffers from important shortcomings: it requires an input parameter that affects both the performance and the accuracy of the results. Thus, different thresholds can be extracted from the same dataset and metric.

3. DISCUSSION:

In this paper, I provided a discussion about threshold extraction methods, which are summarized in Table 1 and 2. I can observed that there are several methods for this purpose. However, there is not a method that is widely recognized by researchers and software engineers as an effective instrument to control the internal quality of software systems. I also observed that using benchmark of systems is an interesting approach, which tends to reflect the software development practice.

Table 1: Thresholds approaches

Authors	Systems	Languages	Metrics
Erni and Lewer- entz [7]	1	Smaltalk	Complexity, coupling, and cohesion
Lanza and Marinesu [14]	82	C++ and Java	Inheritance, coupling, size, and complexity
Alves *et al.* [2]	100	C# and Java	McCabe complexity, unit size, unit interfacing, module interface , size and FAN-IN
Ferreira *et al.* [9]	40	Java	LCOM, DIT, COF, Afferent coupling, NOMP, and NOAP
Shatnawi *et al.* [21]	1	Java	CBO, RFC, WMC, LCOM, DIT, NOC, CTA, CTM, NOAM, NOOM, NOA, and NOO
Catal *et al.* [4]	5	C and C++	SLOC, MCave, EC, DC
Benlarbi *et al.* [3]	2	C++	WMC, DIT, NOC, CBO, and RFC
Herbold *et al.* [8]	8	C, C++, C#, and Java	Size, coupling, complexity, and inheritance
Oliveira *et al.* [16]	86	Java	Size metrics
Oliveira *et al.* [13]	103	Java	Size, coupling, complexity, and cohesion

Table 2: Thresholds approaches

Authors	Method	Weaknesses
Erni and Lewerentz [7] Lanza and Marinescu [14]	mean and standard deviation	It requires an input parameter that affects both the performance and the accuracy of the results
Alves *et al.* [2]	quantile function analysis	The goal is to create quality profiles to rank entities
Ferreira *et al.* [9]	statistical distribution analysis	do not establish the percentage of classes tolerated in each category
Shatnawi et al. [21] Catal et al. [4]	ROC curves	This methodology does not succeed in deriving monotonic thresholds and thresholds values can be not found
Benlarbi *et al.* [3]	linear regression	There is no empirical evidence supporting the model
Herbold *et al.* [8]	machine learning	The methodology produces only a binary classification
Oliveira et al. [16] Oliveira et al. [13]	K-means algorithm	It requires an input parame- ter that affects both the per formance and the accuracy of the results.

4. CONCLUSION:

Source code metrics can be used to find possible problems or chances for improvements in software quality. A variety of metrics to measure source code properties like size, complexity, cohesion, and coupling have been proposed [1, 14]. How- ever, source code metrics are rarely used to support decision making because they are not easy to interpret [85, 95]. To promote the use of metrics as an effective measure- ment instrument, it is essential to establish credible thresholds [2, 21, 8].

REFERENCES

1. Abreu, F. B. and Carapuça, R. (1994). Object-oriented software engineering: Mea- suring and controlling the development process. In 4th International Conference of Software Quality (ICSQ), pages 3–5.
2. Alves, T. L., Ypma, C., and Visser, J. (2010). Deriving metric thresholds from benchmark data. In 26th IEEE International Conference on Software ftaintenance (ICSft), pages 1–10.
3. Benlarbi, S., Emam, K. E., Goel, N., and Rai, S. (2000). Thresholds for object- oriented measures. In 11th International Symposium on Software Reliability Engi- neering (ISSRE), pages 24–38.
4. Catal, C., Alan, O., and Balkan, K. (2011). Class noise detection based on software metrics and ROC curves. Information Sciences, 181(21):4867–4877.
5. Chidamber, S. and Kemerer, C. (1991). Towards a metrics suite for object oriented design. In 6th Conference Proceedings on Object-
6. oriented programming systems, lan- guages, and applications (OOPSLA), pages 197–211.
7. Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6):476–493.
8. Erni, K. and Lewerentz, C. (1996). Applying design-metrics to object-oriented frameworks. In 3rd IEEE International Software metrics Symposium (ftETRICS), pages 64–74.
9. Herbold, S., Grabowski, J., and Waack, S. (2011). Calculation and optimization of thresholds for sets of software metrics. Journal of Empirical Software Engineering, 16(6):812–841.
10. Ferreira, K., Bigonha, M., Bigonha, R., Mendes, L., and Almeida, H. (2011). Identifying thresholds for object-oriented software metrics. Journal of Systems and Software, 85(2):244–257.
11. Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). Refactoring: Improving the Design of Existing Code. Addison Wesley.
12. Kan, S. H. (2002). ftetrics and ftodels in Software Quality Engineering. Addison- Wesley, 2 edition.
13. Kitchenham, B. (2009). What is up with software metrics?-A preliminary mapping study. JournaL of Systems and Software, 83(1):37–51.
14. Oliveira, P., Borges, H., Valente, M. T., and Costa, H. (2013). Metrics-based de- tection of similar software. In 25th International Conference on Software Engineering and Knowledge Engineering (SEKE), pages 447–450.
15. Lanza, M. and Marinescu, R. (2006). Object-Oriented ftetrics in Practice: Using Software ftetrics to Characterize, Evaluate, and Improve the Design of Object- Oriented Systems. Springer.
16. Lehman, M. M. (1996). Blaws of software evolution revisited. In 5th European Workshop on Software Process Technology (EWSPT), pages 108–124.
17. Oliveira, P. M., Borges, H. S., Valente, M. T., and Costa, H. A. X. (2012). Uma abordagem para verificaï£¡ï£¡o de similaridade entre sistemas orientados a objetos. In XI Simpósio Brasileiro de Qualidade de Software (SBQS), pages 1–15.
18. Yoon, K.-A., Kwon, O.-S., and Bae, D.-H. (2007). An approach to outlier de- tection of software measurement data using the k-means clustering method. In 1st Symposium on Empirical Software Engineering and fteasurement (ESEft), pages 443–445.
19. Newman, M. E. J. (2005). Power laws, pareto distributions and zipf's law. Con- temporary Physics, 46(5):223–351.
20. Pressman, R. S. (2009). Software Engineering: A Practitioner's Approach. McGraw-Hill Science, 7 edition.
21. Queiroz, R., Passos, L., Valente, M. T., Apel, S., and Czarnecki, K. (2014). Does feature scattering follow power-law distributions? an investigation of five pre-processor-based systems. In 6th International Workshop on Feature-Oriented Software Development (FOSD), pages 23–29.
22. Shatnawi, R., Li, W., Swain, J., and Newman, T. (2010). Finding software metrics threshold values using ROC curves. Journal of Software ftaintenance and Evolution: Research and Practice, 22(1):1– 16.
23. Vale, G., Albuquerque, D., Figueiredo, E., and Garcia, A. (2015). Defining metric thresholds for software product lines. G. Eason, B. Noble, and I.N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529-551, April 1955.