Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density

Kazuhiro Yamashita*, Changyun Huang*, Meiyappan Nagappan‡, Yasutaka Kamei*,
Audris Mockus§, Ahmed E. Hassan†, and Naoyasu Ubayashi*
* Kyushu University, Japan; {yamashita, huang}@posl.ait.kyushu-u.ac.jp, {kamei, ubayashi}@ait.kyushu-u.ac.jp
† Queen's University, Canada; ahmed@cs.queensu.ca
‡ Rochester Institute of Technology, USA; mei@se.rit.edu
§ University of Tennessee-Knoxville, USA; audris@utk.edu

***Abstract*—Practical guidelines on what code has better quality are in great demand. For example, it is reasonable to expect the most complex code to be buggy. Structuring code into reasonably sized files and classes also appears to be prudent. Many attempts to determine (or declare) risk thresholds for various code metrics have been made. In this paper we want to examine the applicability of such thresholds. Hence, we replicate a recently published technique for calculating metric thresholds to determine high-risk files based on code size (LOC and number of methods), and complexity (cyclomatic complexity and module interface coupling) using a very large set of open and closed source projects written primarily in Java. We relate the threshold-derived risk to (a) the probability that a file would have a defect, and (b) the defect density of the files in the high-risk group. We find that the probability of a file having a defect is higher in the very high-risk group with a few exceptions. This is particularly pronounced when using size thresholds. Surprisingly, the defect density was uniformly lower in the very high-risk group of files. Our results suggest that, as expected, less code is associated with fewer defects. However, the same amount of code in large and complex files was associated with fewer defects than when located in smaller and less complex files. Hence we conclude that risk thresholds for size and complexity metrics have to be used with caution if at all. Our findings have immediate practical implications: the redistribution of Java code into smaller and less complex files may be counterproductive.**

***Index Terms*—Software metrics; Thresholds; Defect models;**

I. Introduction

There has been a considerable amount of research in the field of software defect prediction, especially in the last decade. Over 100 papers have been published just from 2000 - 2011 in this area of Empirical Software Engineering (ESE) research alone [16, 27]. The primary goal of such research is to be able to provide guidelines to practitioners on what kind of code has better quality. This goal has two purposes: (a) identifying the metrics that are most useful for the practitioner (often usefulness is measured by how accurately the metrics identify bad code, how easy is it to collect the metrics, and how actionable the metrics are), and (b) what values of these metrics indicate good code or bad code.

We focus on the second purpose. In particular, over the last 30 years, many publications investigated "best" values for the various software metrics. Among other methods, thresholds were derived based on experience [8, 23, 25], analysis of the metrics themselves [11, 14], error models [6, 10, 26], cluster techniques [22, 34], and metric distributions [31, 32].

Almost all of these approaches treat extreme values of file metrics as detrimental. Defect prediction approaches that use a linear statistical model, conclude that an extreme value of the metrics implies poor quality of code, by virtue of choosing such a model. For example, 'larger files will have more defects' is a common refrain in ESE research [15]. Alternatively the "Goldilocks Principle" suggests that extreme values of a metric are a sign of poor quality code [17, 18]. Fenton and Neil provide a more comprehensive list of research studies with respect to metrics based defect prediction [12]. They find that the literature has contradictory evidence regarding the relationship between software defects and software metrics like size and complexity. It is, thus, unclear if metric thresholds should be used to identify source code files that are at high risk.

Consequently, we aim to observe *if a consistent relationship between metric thresholds and software quality* is present in OSS and industrial projects. Therefore we conduct a case study on three OSS and four industrial projects. The metrics that we choose to evaluate are size based (Total LOC in a file, and Module interface size in a file), and complexity based (cyclomatic complexity and module inward coupling). We evaluate software quality using two criteria - (a) defect proneness (probability of a file having a defect), and (b) defect density (the number of defects/LOC).

We replicate the most recently published state-of-the-art technique (proposed by Alves *et al.* [3]) to determine the thresholds for these metrics, and found them to be very close to ones reported earlier. In order to use this approach, we need a set of projects to calculate the thresholds from. In their paper Alves *et al.* [3] use 100 Java and C# systems. In our paper we use 1,000 OSS Java projects (from a pool of more than 20K OSS Java projects) to calculate the thresholds for the OSS projects and 200+ industrial systems to calculate the thresholds for the industrial projects. Our results would be similar with alternative, simpler approaches to set the thresholds, but we prefer to use the state-of-the-art technique to ensure that the thresholds are set in the most appropriate manner. We evaluate software quality using two criteria - (a) defect proneness (probability of a file having a defect), and

(b) defect density (the number of defects/LOC). For reasons listed in Section II-A, we consider Java projects in this case study, and thus our findings relates only to Java files.

On replicating the approach proposed by Alves *et al.* [3] for determining thresholds in our case study, we make the following key observations:

- Files identified as very high-risk by size and complexity thresholds were associated highest fault-pronenes.
- Files identified as very high-risk by size threshold were several times more fault-prone than files identified as very high-risk by complexity threshold.
- Files identified as very high-risk by size threshold have a lower (often significantly lower) defect density than the remaining files.
- In some cases files identified as very low risk by size and complexity threshold had a low defect density value as well. In these cases the files with medium to high risk size and complexity values had the highest defect density (which is contradictory to the "Goldilocks Rule").

These findings confirm earlier results that the size benchmarks are associated with high defect-proneness. They also indicate that the size is a better threshold of defect-proneness than complexity. Defect density (in contrast to defect-proneness) tends to be the lowest in the files benchmarked as high-risk, yet we could not confirm earlier suggestions that the smallest files always have the highest fault density or the "Goldilocks rule". *In summary, we found that the thresholds have to be used with caution, if at all.*

Paper organization. Follows: Section II describes the data that we use in our case study. Section III presents the technique that we use to determine the thresholds for the metrics. Section IV presents the design of our experiments. Section V presents the results of our experiments and the discussion about the results. In Section VI, we present the related research on ways to determine the thresholds for metrics, and on the veracity of metric thresholds. Section VII presents the threats to validity, and finally Section VIII presents our conclusions.

II. Data Sources

In this case study, we analyzed both OSS and industrial Java software projects. We describe the data that we used in our case study below. This section is further subdivided into two subsections - one for describing the projects that were used to calculate the thresholds, and one for describing the projects that we used to determine the relationship between thresholds and defects.

A. Software Projects used for Calculating Thresholds

In the case of the industrial projects we calculated the thresholds from 205 industrial Java software projects also from Avaya. In the case of the OSS projects we use the dataset collected by Mockus [24]. Mockus created this dataset from several forges such as git.kernel.org, SourceForge, PostgreSQL, GitHub and so on. This dataset contains more than 125K software projects (in C, C++, C#, Java etc.). We extracted 22,956 Java projects from this dataset. We chose to only extract Java projects for 2 reasons - (a) A lot of open source and industry projects are written in Java, and (b) We wanted to control for language context, as Java may have some specific effects on file size. The use of a single language excludes the possibility that our results were affected by the thresholds potentially having opposite effects in different languages. We only included those Java projects that had 200 or more files in our case study to exclude very small projects. Other investigations we have done (beyond the scope of this paper) suggest that a much higher percent of projects in the forges with fewer than 200 files appear to be not real software development projects. Therefore, we were left with 4,575 Java projects in the end.

B. Software Projects with Defect Data

Although we have this large set of projects to calculate the thresholds, we do not have defect data for them. Hence, we collected defect data in three OSS and four industrial (from a telecommunications company, Avaya) Java software projects. The three OSS projects are - *Eclipse, Mylyn, and Netbeans*. Eclipse and Netbeans are IDEs for writing Java code, while Mylyn is a plugin for Eclipse used in task management. All three projects have been used in many software engineering studies before [7, 19, 30]. We collected this data for three versions in each of the OSS projects. We collect three versions of these OSS projects even though all our metrics (size and complexity) are non-process based (product-based), and hence static in nature. We do this in order to examine if our results are consistent across multiple versions of the same project.

The industrial projects are from a telecommunications company and are products that are being used regularly. We only have one version of each of these projects. Therefore, in total we have 13 versions ($3 * 3 = 9$ OSS and $1 * 4 = 4$ industrial) of software projects for which we collected the defect and metric data. We collected the defect data for these projects at a file level. Each commit to a file with a commit message that contains an issue ID that exists in the issue tracking system (note that in the industrial projects, we only consider customer-reported defects) is considered to be a defect that was fixed in that file. This is the same procedure used by Kamei *et al.* [19]. Note that in this paper, we use post release defects, which are found after releasing a version of software. They are reported in the issue tracking system (such as Bugzilla). Post release defects could be anything from a typo to serious security/performance defects or runtime errors/crashes. Currently in all software engineering research all post release defects are considered to be harmful since they affect the customer.

III. Deriving Thresholds for Metrics

As explained in the previous section, we collected 4,575 OSS projects and 205 industrial projects for determining the thresholds for the 3 OSS and 4 industrial projects with defect data. In this section we explain the method that we replicated to determine the thresholds [3] and apply this technique to all the 4,575 OSS projects to give an overview of the OSS

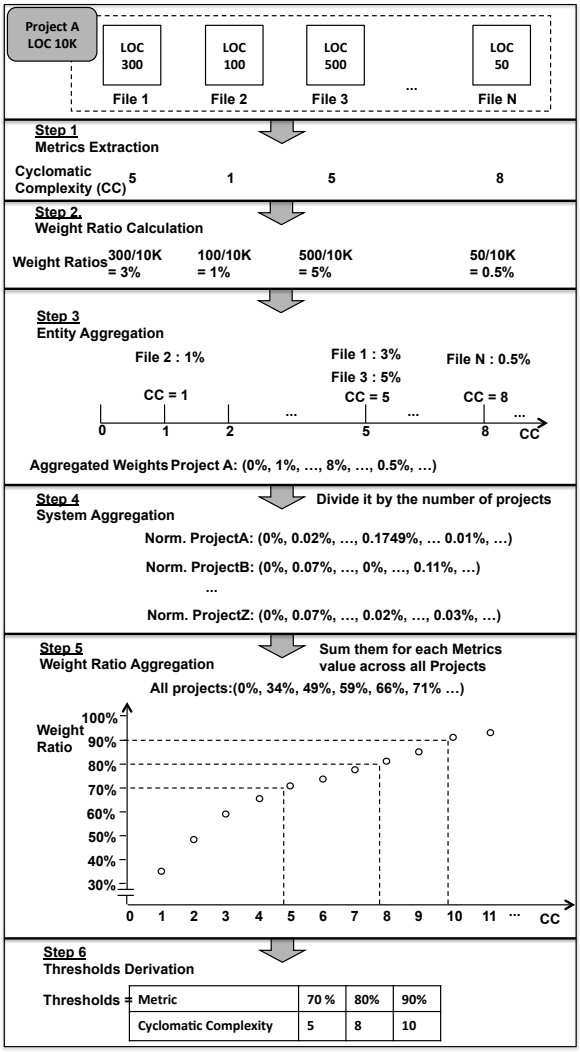


Fig. 1. The six steps in the technique proposed by Alves *et al.* [3] to derive the thresholds.

dataset being used. We chose to replicate this technique for calculating the thresholds for a number of reasons: (a) it adheres to the scale and distribution of the metrics themselves, (b) the inherent variability present in the various metrics across the various systems are brought out, and (c) the thresholds derived have an empirical foundation and do not depend on the experiences of an individual,

Figure 1 illustrates the technique used by Alves *et al.* which consists of the following six steps [3]:

1) Metrics extraction: We extract the four metrics (lines of code, module interface size, cyclomatic complexity, and module interface coupling) for each file in each of the software projects under study using the tools, Understand [2], and Sonar [1]. For example as shown in Step 1 of Figure 1, say Project A has files A_1, A_2, ..., A_n. We extract the four metrics for all the n files.

2) Weight ratio calculation: We calculate the weight ratio of each file in each system as the ratio of the size of the file to the total size of the system. For example as shown in Step 2 of Figure 1, if file A_1 has 300 LOC, and if the total lines of code of project A is 10K LOC, then the weight ratio for file A_1 will be 300/10K = 3%. Thus there is one weight ratio for each file, in each project.

3) Entity aggregation: In this step, we aggregate the weight ratios of each file according to the metric value. For example as shown in Step 3 of Figure 1, if files A_1, and A_3 are the only files in project A with a cyclomatic complexity of 5, then we add the weight ratios corresponding to files A_1 and A_3. If the weight ratios of A_1 and A_3 are 3% and 5% respectively, then the entity aggregated value for cyclomatic complexity of 5 is 8% (3 + 5). Therefore we calculate this aggregated value for every unique value of all the four metrics.

4) System aggregation: In this step, we take the entity aggregated value of each unique metric value for all the four metrics from the previous step, and divide each by the total number of projects (in our case, this is 4,575), to get the normalized entity aggregation value. Thus the normalized entity aggregation value in our example for cyclomatic complexity value of 5 for project A is 0.1749%, as shown in Step 4 of Figure 1. Then we aggregate these values across all the projects. Therefore in our example we would add the normalized aggregation value for a cyclomatic complexity value of 5 from all 4,575 projects. We would repeat this for every unique value of all the four metrics across all the projects. In the end we have four vectors, one for each metric. The length of each metric is equal to the number of unique values that the metric can assume across the 4,575 projects. Thus we have aggregated the metrics across the 4,575 projects.

5) Weight ratio aggregation: In this step, we take the four vectors generated at the end of the previos step, sort them in ascending order. Then we plot a cumulative line chart for each of the metrics (in percentile scale). This is shown in Step 5 of Figure 1.

6) Thresholds derivation: In the final step we arrive at the thresholds. As noted by Alves *et al.* in their paper, we too determine the value of each of the metrics at the $70\%^{th}$, $80\%^{th}$, and $90\%^{th}$ percentiles. We use these values as the thresholds for various risk categories of the metrics. For example as shown in Step 5 of Figure 1, if the cyclomatic complexity thresholds are 5, 8, and 10 (corresponding to the $70\%^{th}$, $80\%^{th}$, and $90\%^{th}$ percentile values), then we classify all files with cyclomatic complexity, $cc < 5$ as *Low Risk*, $5 <= cc < 8$ as *Medium Risk*, $8 <= cc < 10$ as *High Risk*, $cc >= 10$ as *Very High Risk*. Therefore each file will be assigned to exactly one risk category based on the metric value for that file. Note that a particular file could be in the low risk category from the perspective of one metric, and in a high risk category from the perspective of another metric. For example, file A_1 which has a cyclomatic complexity of 5 and 300 LOC, will be classified as a medium risk file with respect to cyclomatic complexity and a very high risk file with respect to LOC. For further details on this approach for calculating thresholds we refer the readers to the original paper by Alves *et al.* [3].

TABLE I
CORRESPONDENCE BETWEEN THE METRICS ANALYZED AND THE METRICS IN THE TOOLS: UNDERSTAND AND SONAR

Metric	Granularity	Understand	Sonar
Lines of Code	File	CountLineCode	ncloc
Module Interface Size	File	CountDeclInstanceMethod	functions
Cyclomatic Complexity	Method	CyclomaticStrict	function_complexity
Module Inward Coupling	Method/File	CountInput	accessors

IV. CASE STUDY DESIGN

In the previous sections we have described our data sources, and the technique we use to derive the thresholds for the projects, along with how we determine the number of defects in each file of each version of the three OSS and four industrial projects. In this section, we present the study design of our experiments to evaluate the relationship between metric thresholds and software quality. The details of the metrics used in this study, and the evaluation criteria to quantify software quality are described below.

A. Metrics that were Analyzed

In this case study we study two size based and two complexity based metrics. We chose these four metrics, since Alves *et al.* [3], in their original case study, used these four metrics and derived the thresholds on 100 OSS and industrial projects. These metrics were extracted from the OSS projects using the tool Understand [2], and from the industrial projects using the tool Sonar [1]. The metrics analyzed and their corresponding functions in the tools are presented in Table I. Additional details on these metrics and how they are calculated can be found at the webpages for the corresponding tools [1, 2]. The two size based metrics were collected at the file level, while the two complexity based metrics was collected at the method level. Since Sonar, the tool that was used to extract the metrics for the industrial project, did not collect the Module Inward Coupling at a method level, we used the number of accessors as a substitute which is measured at the file level. The threat which is caused by using two different tools to measure metrics is discussed in Section VII.

Table II shows the statistics of OSS and Avaya projects with respect to four analyzed metrics. We first calculate median values of four metrics shown in Table I for each OSS and Avaya project. Then we calculate minimum, median and maximum values to lift up project-level values to category-level ones (i.e., OSS-level or Industry-level).

B. Software Quality Evaluation Criteria

We use two well known software quality measures in order to evaluate our goal - defect proneness and defect density [27]. Typically these are defined for a specific file or class. In our case, the granularity that we examining them in are: risk categories. Note that these risk categories are defined by the thresholds that we extract for the metrics. We define the software quality evaluation criteria as follows:

1) Defect proneness - the probability of a file in a particular risk category having a defect. More formally, it is the ratio of the number of files in a risk category which has at least one defect, to the number of files present in that category. Let's assume that there are 400 files that have a cyclomatic complexity value less than the value for the 70^{th} percentile (low risk) in our threshold derivation technique. If 100 of them have at least one defect, then the defect proneness of the low risk category of files from the perspective of cyclomatic complexity is $100/400 = 0.25$.
2) Defect density - the number of defects per LOC for a particular risk category. More formally, it is the ratio of the sum of the defects in all the files in a particular category, to the sum of LOC of all the files in the same category. Considering the above example, if the same 400 files have a combined size of 45,000 LOC, and 50 of the 100 files with defects have 2 defects each (which makes it $50 * 1 + 50 * 2 = 150$ defects in total), then the defect density of files with low risk from the perspective of cyclomatic complexity is $150/45,000 = 0.0033$.

Therefore we calculate the defect proneness, and defect density of each risk category (four in total), for all the metrics (four in total) in all the case study projects (three OSS and four industrial, and 13 in total). As we mentioned in Section II, in our case study we use post release defects to calculate both of these quality measures. In the case of OSS, these defects were reported in the bug repository, and in the case of the industrial projects, the defects were customer reported.

C. Choosing Representative Projects for Calculating Metrics Thresholds

One of the assumptions made by Alves *et al.* is that the thresholds are derived from a set of systems representative of the target system to obtain robustness of the derived threshold. Note that the case study systems that we calculate the quality measures are not the same as the case study systems that we use to calculate the thresholds. We calculate the quality measures for three OSS and four industrial projects. These projects are the target systems in our case. To make sure that the thresholds are calculated for the target systems from a more representative sample, we chose a subset of 1,000 projects from the 4,575 Java projects for each of the three versions of the three OSS projects. By "representative" we mean having a similar distribution of file sizes. For selection of the most similar projects we use median to characterize that distribution. More discussion is in Section VII. The selection was done as follows:

1) For all of the nine versions of OSS projects and 4,575 projects used to determine the thresholds, we calculated the median LOC in a file.
2) For each of the nine versions of OSS projects, we chose 1,000 projects from the pool of 4,575 projects that had the closest median LOC in a file.

TABLE II
STATISTICS OF OSS AND AVAYA PROJECTS WITH RESPECT TO FOUR METRICS

	Lines of Code			Module Interface Size			Cyclomatic Complexity			Module Inward Coupling		
	MIN	MED	MAX	MIN	MED	MAX	MIN	MED	MAX	MIN	MED	MAX
OSS	2.0	29.0	403	0.0	3.0	26.0	1.0	1.0	5.0	0.0	2.0	6.0
Avaya	5.0	41.5	161.0	1.0	4.0	11.0	1.0	6.5	32.0	0.0	0.0	4.0

TABLE III
MEDIAN OF THE OSS PROJECT, ALONG WITH THE INTER QUARTILE RANGE OF THE DIFFERENCE BETWEEN MEDIAN LOC/FILE OF EACH OF THE 1,000 OSS PROJECTS CHOSEN TO DERIVE THE THRESHOLD AND THE MEDIAN LOC/FILE IN THE CORRESPONDING OSS PROJECT.

Project Name	Median	Interquartile Range
Eclipse 3.0	100	8
Eclipse 3.1	105	8
Eclipse 3.2	105	8
Mylyn 1.0	93	6
Mylyn 2.0	89	5
Mylyn 3.0	93	6
Netbeans 4.0	117	11
Netbeans 5.0	119	11
Netbeans 5.5.1	106	9

In Table III, we present the median LOC/File of each of the nine OSS projects. These are the target projects for which we need to calculate the threshold. As stated above, we intend to choose 1,000 projects that are most representative of the target systems in order to calculate the thresholds. Therefore we calculate the Inter Quartile Range (IQR) of the absolute difference between the median LOC/File in the 1,000 projects chosen for deriving the thresholds and the median LOC/File of the target project, in order to examine how representative the sample used to calculate the threshold is to the target project. IQR is a robust measure of spread that is more appropriate than the variance or standard deviation for the highly skewed distribution such as file size. For all the target systems the IQR of the difference between their median LOC/File and the median LOC/File in the 1,000 projects chosen to calculate the threshold is at-most 11 LOC. From this low value we can see that the 1,000 projects chosen from the dataset of 4,575 projects, to derive the metric thresholds, are indeed comparable in median LOC/File value to each of the OSS projects under study.

Using the size and complexity metrics extracted from the 1,000 projects for each target project, we calculated a unique set of thresholds for each of the nine versions of OSS projects. As for the four industrial projects we used the set of 205 different industrial projects from the same organization to derive the threshold (i.e., we use same threshold to evaluate four industrial projects). The thresholds for the nine OSS projects and the thresholds for the industrial projects are presented in Table IV. From the four tables, we can see that the industrial projects have a considerably higher range for the thresholds of all the metrics. The 70^{th} percentile values are much lower than the OSS projects, and the 90^{th} percentile values are much higher than their OSS counterparts.

V. RESULTS AND DISCUSSION

In this section, we present the experiment that we carried out along with the results for those experiments and a discussion pertaining to it.

A. Experimental Approach

As described in the previous section, we extract the metrics for all the OSS and industrial projects. We derive the thresholds individually for each OSS software version and collectively for the industrial projects. Using the thresholds for each metrics, we split the files in each of the three OSS and four industrial projects (13 versions in total), for which we also have the defect data, into the four risk categories(low, medium, high and very high). Using this classification and the defect data, we calculate the defect proneness, and defect density for each risk category of each metric for the 13 versions.

B. Experimental Results

The defect proneness and defect densities are shown as line plots in Figure 2 and 3. The plot on the left hand side corresponds to the defect proneness, and the plot to the right hand side corresponds to the defect density of each system. Overall, most of the trends among all the systems are similar. All the industrial projects bear close resemblance to each other. Additionally there is not much variance among versions of each OSS project either. Some observations that we can make on more careful analysis from Figure 2 and 3 are as follows:

(B1) Defect Proneness:

1) In the industrial projects, files in the low risk category have the same defect proneness, irrespective of the metric that was used to assign the risk, i.e., files that have few LOC and low complexity are equally defect prone. However, files in the high risk category have considerably different defect proneness values. For example, files that have a very high value for size based metrics are two to five times more defect prone than files that have very high values for complexity based metrics. When we examine the OSS projects, we consistently find that files are twice as likely to be defect prone for both low and high values of the size based metrics, than the corresponding low and high values of the complexity based metrics.
2) In the OSS projects, in some cases (like Eclipse 3.0, Eclipse 3.1, and Netbeans 4.0), the highest risk category of size based metrics are not related to defect proneness as expected. The files with very high values for the sized based metrics are marginally less defect prone than files

TABLE IV
THRESHOLDS FOR THE OSS AND INDUSTRIAL PROJECTS

(a) Lines of Code

Project Name	Version	70%	80%	90%
Eclipse	3.0	335	498	884
Eclipse	3.1	346	515	908
Eclipse	3.2	346	515	908
Mylyn	1.0	321	480	861
Mylyn	2.0	313	466	847
Mylyn	3.0	321	480	861
Netbeans	4.0	365	545	952
Netbeans	5.0	369	553	975
Netbeans	5.5.1	346	514	911
Industrial	-	324	543	1,423

(b) Module Interface Size

Project Name	Version	70%	80%	90%
Eclipse	3.0	18	26	43
Eclipse	3.1	19	27	44
Eclipse	3.2	19	27	44
Mylyn	1.0	18	26	42
Mylyn	2.0	18	25	42
Mylyn	3.0	18	26	42
Netbeans	4.0	20	28	46
Netbeans	5.0	20	28	47
Netbeans	5.5.1	19	27	44
Industrial	-	19	30	89

(c) Cyclomatic Complexity

Project Name	Version	70%	80%	90%
Eclipse	3.0	6	9	17
Eclipse	3.1	6	9	18
Eclipse	3.2	6	9	18
Mylyn	1.0	6	9	17
Mylyn	2.0	6	9	16
Mylyn	3.0	6	9	17
Netbeans	4.0	6	10	18
Netbeans	5.0	6	10	19
Netbeans	5.5.1	6	9	17
Industrial	-	4.3	5.6	10.3

(d) Module Inward Coupling

Project Name	Version	70%	80%	90%
Eclipse	3.0	6	9	14
Eclipse	3.1	7	9	14
Eclipse	3.2	7	9	14
Mylyn	1.0	6	9	13
Mylyn	2.0	6	9	13
Mylyn	3.0	6	9	13
Netbeans	4.0	7	9	15
Netbeans	5.0	7	9	15
Netbeans	5.5.1	7	9	14
Industrial	-	3	7	30

with just high values for the same metrics. However, in the industrial projects, this phenomenon is observed for the cyclomatic complexity instead. Thus in these cases, we observe that there is no monotone relationship between these metrics and defect proneness.

(B2) Defect Density:

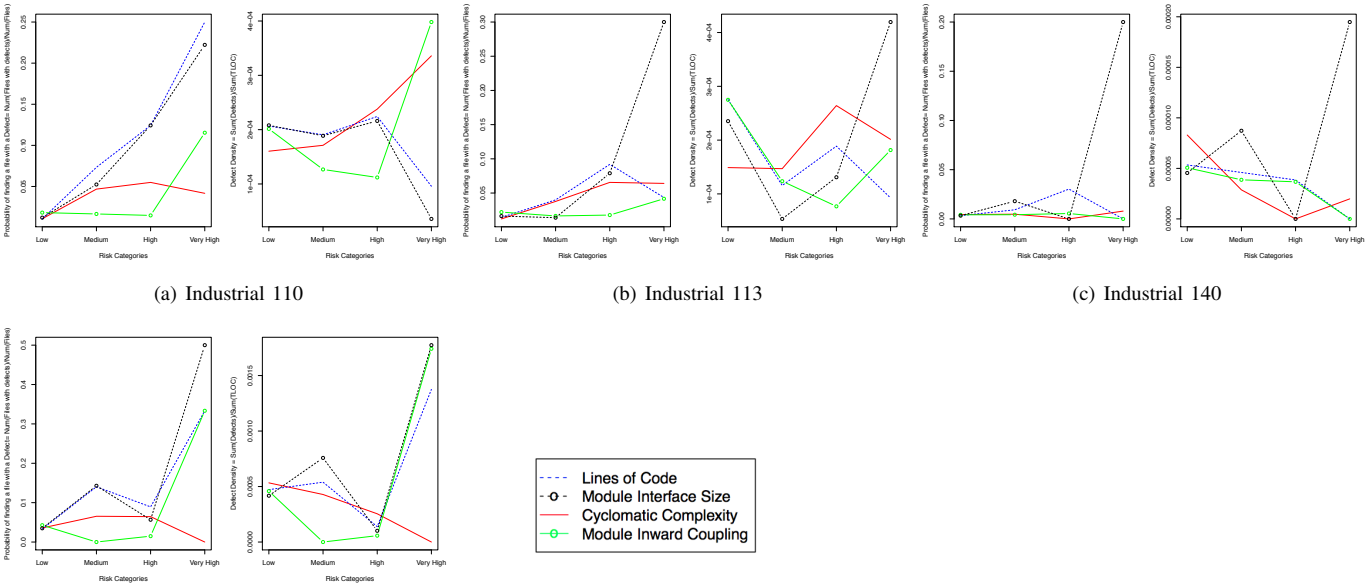
1) In all three version of Eclipse and Mylyn, we can observe that the relationship between size based metrics and defect density, is a monotonically decreasing one. The higher the risk category that a file is in with respect to size based metrics, the lower is its defect density. However, when we look at the complexity based metrics for these two projects, we observe that there is a inverted-U shaped curve. This indicates that the files with medium values for the complexity based metrics, have a higher defect density than files with extreme complexity values.
2) When looking at Netbeans, we can see that in version 5.0 and 5.5.1, the relationship between the various metrics and defect density is all over the place. There is no consistent trend. However, in version 4.0, we can clearly see that except for cyclomatic complexity, the other three metrics have a rotated-Z curve. We can see that the defect density of the files in the very high risk category are slightly higher than the defect density of files in the medium risk category. However, as observed in Eclipse and Mylyn, the defect density of the files in the very high risk category (the highest), is much lower than the defect densities in both the medium and high risk categories.
3) In the industrial systems, we see that the files with the highest value for module interface size (accessors in this case) are also the files with the highest defect density (except Industrial 110). Also except in Industrial 208, the files with the largest value for LOC seems to have the lowest defect density.

As seen in the result of this subsection, *the relationship between defect density and metric threshold values are not consistent*. To quantify if the defect density is indeed higher for the very high risk values of each metric, we model code defects via a poisson model that includes two predictors (independent variables): the amount of code in the set of files within a risk class and an indicator function for the highest risk threshold. The response (dependent variable) is the number of defects observed for that set of files. The fitted coefficient at the very high risk indicator would allow us to determine if the files in the highest risk categories are indeed more defect prone than other files. We present the results in Table V. In the table, z value is estimated coefficient value divided by the value of standard error and $Pr(> |z|)$ shows significance of the z value.

From the Table V we can see that in almost all the projects and for all the metrics the z value is negative, indicating that the files in the highest risk category for all the metrics have indeed lower defect density. However, only in the case of LOC and cyclomatic complexity (and only in OSS projects), are these results statistically significant. In the rest of the cases, there is no statistical evidence to show the inverse relationship between the metrics and defect density.

(B3) Summary:

From Subsection V-B, we can see that defect proneness of files in a category has an almost linear relationship to the values that correspond to that risk category. Hence this validates the bulk of the existing literature on defect prediction. However, when looking at the observations made in



(a) Industrial 110 (b) Industrial 113 (c) Industrial 140

(d) Industrial 208

Fig. 2. Defect Proneness and Defect Density of each risk category in each case study system

TABLE V
z AND $Pr(>|z|)$ OF THE INTERCEPT OF THE POISSON MODEL. VALUES WITH * NEXT TO THEM INDICATE STATISTICAL SIGNIFICANCE WITH $p < 0.05$

	Lines of Code		Module Interface Size		Cyclomatic Complexity		Module Inward Coupling	
	z	$Pr(>\|z\|)$	z	$Pr(>\|z\|)$	z	$Pr(>\|z\|)$	z	$Pr(>\|z\|)$
Eclipse 3.0	-9.56	1.19e-21*	-5.16	2.50e-07*	-7.14	9.05e-13*	-3.70	0.02e-02*
Eclipse 3.1	-15.31	6.10e-53*	-8.90	5.61e-19*	-9.41	4.96e-21*	-7.37	1.72e-13*
Eclipse 3.2	-7.001	2.53e-12*	-0.52	0.60	-8.11	5.02e-16*	-6.63	3.40e-11*
Mylyn 1.0	-2.48	0.013*	-0.24	0.81	-4.12	3.67e-05*	-2.10	0.04*
Mylyn 2.0	-1.20	0.23	-0.24	0.81	-2.95	0.03e-01*	-0.88	0.38
Mylyn 3.0	-2.33	0.02*	-4.46	8.23e-06*	-2.98	0.03e-01*	-1.13	0.26
Netbeans 4.0	-3.78	0.02e-02*	-1.56	0.12	0.34	0.73	-2.28	0.02*
Netbeans 5.0	-2.09	0.04*	-0.43	0.67	-1.88	0.06	-0.66	0.51
Netbeans 5.5.1	-4.10	4.15e-05*	-0.26	0.80	-2.90	0.04e-01*	-1.18	0.24
Industrial 110	-2.19	0.03*	-2.88	0.04e-01*	-1.32	0.19	1.45	0.15
Industrial 113	-1.26	0.21	2.33	0.02*	0.25	0.81	1.91	0.06
Industrial 140	-0.05e-02	0.99	-0.06	0.95	0.34	0.73	-0.05e-02	0.99
Industrial 208	1.05	0.30	1.93	0.05	-0.09e-02	0.99	2.85	0.04e-01*

Subsection V-B, we can conclude that in almost all cases the files in the highest risk category have the lowest defect densities. Additionally in many cases, the files in the lowest risk category also have low defect densities. This observation is in contradiction to the "Goldilocks Conjecture", which states that it is ideal for files to not have metric values in either extremes.

VI. Related Work

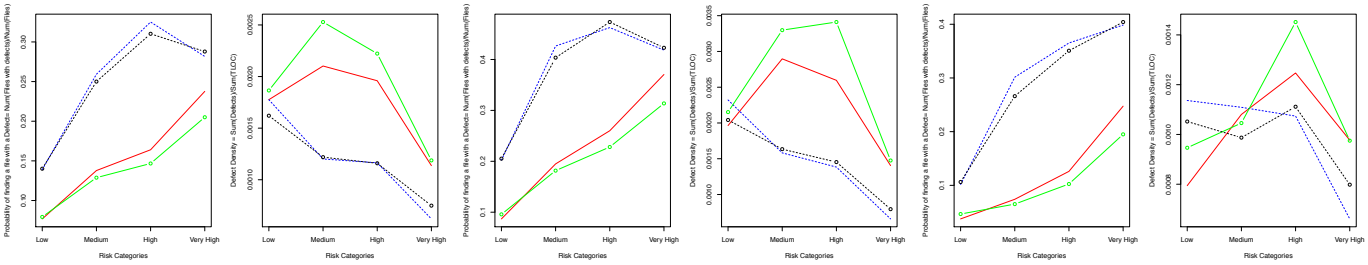
In this paper, we derive thresholds using the technique proposed by Alves *et al.* [3]. More details about this was presented in Section III. However, in this section, we present some of the other research related to thresholds as well as some of the research that questions such thresholds.

A. Techniques for Deriving Thresholds

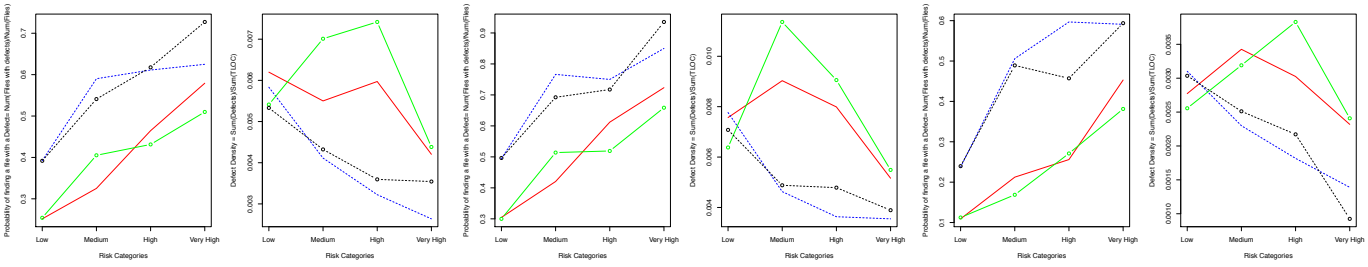
In software engineering research there have been many attempts to define thresholds for various metrics. The rationale for doing so is to provide guidelines for practitioners to identify poor quality code in their software. Such thresholds can be derived based on theory, experience, or empirical analysis of metrics.

Researchers have argued about fault densities from a theoretical perspective [10, 17, 18]. Emam *et al.* theoretically show that smallest files having the largest defect densities are a mathematical artifact. Hatton argues that the software components that "fit best into human short-term memory cache" will produce the lowest defect densities [17, 18]. Hence, both small and large files will tend to have higher defect densities.

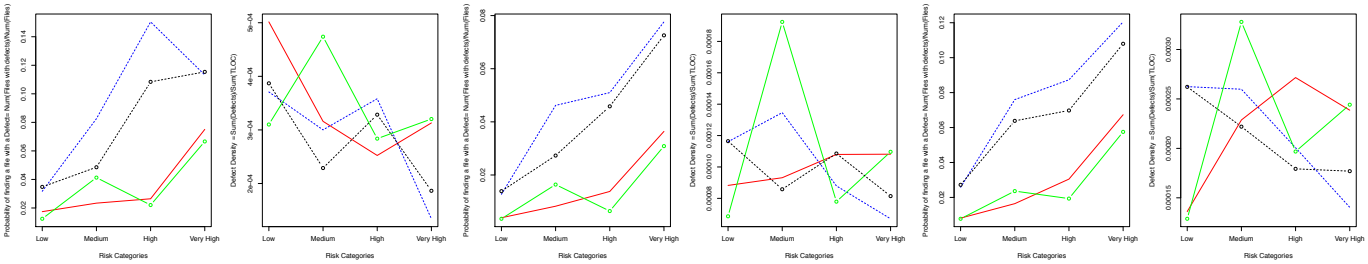
Some of authors defined thresholds by their experience. McCabe [23] defined cyclomatic complexity and recommended the threshold for the McCabe complexity metric to be 10 based on his experience. Nejmeh [25] defined a metric called NPATH, an objective measure of software complexity, which counts the number of acyclic execution paths through a function. Nejmeh defined a threshold of 200 for the NPATH



(a) Eclipse 3.0 (b) Eclipse 3.1 (c) Eclipse 3.2



(d) Mylyn 1.0 (e) Mylyn 2.0 (f) Mylyn 3.0



(g) Netbeans 4.0 (h) Netbeans 5.0 (i) Netbeans 5.5.1

Fig. 3. (contd.) Defect Proneness and Defect Density of each risk category in each case study system

metric based on his experience [25]. Coleman *et al.* [8] defined the Maintainability Index (MI) metric and recommended thresholds of 65 and 85. This implies that when software components with MI values lower than 65, and higher than 85 are difficult to maintain, and between 65 and 85 are moderately maintainable.

Some other authors have empirically evaluated thresholds for the software metrics. For example, Erni *et al.* [11] defined thresholds by using statistical techniques. Their thresholds T are calculated by using mean (μ) and standard deviation (σ) as $T = \mu + \sigma$ or $T = \mu - \sigma$. This technique was evaluated on one system which had three version on metrics such as number of methods in a class, and the lack of cohesion in a method. Shatnawi *et al.* [26] derived thresholds of 12 metrics proposed by Chidamber and Kemerer, Li, and Lorenz and Kidd, by empirically relating the metrics to error severity categories. These thresholds were derived using the receiver operating characteristic (ROC) curves.

In this paper, we use the technique proposed by Alves *et al.* since this is the only technique that takes the statistical distribution inherently present in the data into account [3].

B. Questioning Thresholds

There has also be a few studies over the last three decades that have questioned the veracity of certain thresholds for metrics, as well as the veracity of the concept of thresholds themselves. Basili *et al.* [4], Compton and Withrow [9], and Hatton [17, 18], all questioned the well accepted notion that extensive modularization will reduce defects in the software. They found evidence to the contrary. Their finding suggested that small files in a software had a higher defect density than medium sized files. This new observation is often called the "Goldilocks Conjecture" - files with atypical size are more likely to have more defects than files of typical (medium) size. Additionally, Bell *et al.* [5] found that small files (less than 65 LOC) had high defect densities and that the expected number of defects increased as the square root of the size metric (instead of a linear increase). In our study, we find that large and complex files have lower defect densities, which is contrary to the findings in the studies above. Additonally we also find that in some cases, files with smaller size and complexity as well has lower defect densities. This is exactly the opposite of the "Goldilocks Conjecture" and is contrary to

the findings in the above studies.

Fenton and Neil [12], in their study of the literature on defect prediction approaches, also found that there is research evidence both supporting and refuting the relationship between risk thresholds and size and complexity metrics. However, in their study they do not empirically evaluate the contradictory behaviour. They rely completely on past research. Although, it is an important first step, such a comparison has a limitation - the study design in each the past research studies could be very different. Even they comment on the accuracy of the data collection process in each of the past studies. Therefore the contradictory results could be due to the different study designs and not due to the actual relationship between risk thresholds and metrics. In our paper, we consistently use the same process (described in Sections III and IV), to analyze all the datasets. Hence we are able to empirically verify if risk thresholds have a consistent relationship to size and complexity metrics.

Emam *et al.* questioned if any kind of thresholds for software size was reasonable [10]. They empirically show that there is no statistical evidence to prove that files beyond a certain threshold for size was indeed more defect prone. In the following years, the same authors found move evidence that the size of a class and defect proneness had a power law relationship [20, 21].

Our approach differs from Emam *et al.* [10], and Koru *et al.* [20, 21] in the following ways:

- In addition to software size based thresholds, we also look at complexity based thresholds.
- In addition to examining the relationship to defect proneness we also investigate the relationship to defect density as well.
- Instead of asking the question: is there a threshold for size above which files are more defect prone, we ask: are files benchmarked as high-risk indeed the most defect prone.
- Instead of obtaining a threshold after determining the relationship between quality and size for the whole dataset, we replicate a latest threshold-setting technique.

In summary, the focus of the past studies have been to examine the relationship between a metric like size and quality. Whereas, in our study we focus on determining if thresholds for metrics can reliably identify highly-defect prone and high-defect-density files.

From the perspective of results and conclusions, Koru *et al.* [20, 21] in their work found that, the defect proneness does increase with the size of the software component, but that the rate of this increase slows down (hence having a sub linear power law relationship). Therefore they indirectly come to the conclusion that large files have lower defect density (indirectly because only defect proneness is examined and not defect density). We arrive at a similar conclusion as them. However, we directly relate defect density to size and complexity metrics. Hence, this result from our case study reinforces the conclusions of Emam *et al.* [10], and Koru *et al.* [20, 21]. Additionally we find that in some cases, files with small values for the four metrics also have low defect density. It is the files with medium values for the various metrics that have the highest defect density.

There are other types of thresholds, apart from ones we consider here. For example, code smells [13] are particular patterns that are considered to reflect bad design. Sjøberg *et al.* [29] investigated if smell thresholds for individual files were related to maintenance effort spent on these files and found that there was no relatiobnship if the size of file (LOC) was taken into account. The results we obtained (even though we did not consider several thresholds simultaneously) appear to be consistent with that finding: the high size risk factor was more strongly related to fault-proneness than high complexity risk factor. Other work by the same group [28], found that only the overall size of the system was associated with maintenance effort: the smallest system (among four systems implementing equivalent functionality) was the easiest to maintain despite having much of its code in a single "God" class (one of the code smells evaluated in that work).

VII. Threats to Validity

External Validity: We evaluated our goal with Java projects. Hence, the results may not generalize to projects in different programming languages. To make the study as broad as possible, we examine both OSS (3 versions each of 3 projects) and industrial (four) projects. Additionally for the thresholds we examined 4,575 OSS Java projects. Although, the generalizability of this study can further be improved on repeating it on many more projects, the current size of the case study is the typical size in software defect related literature [27].

Also, in this study we only examine the size and complexity metrics, and as such all our conclusions are limited to them. Further studies on other metrics like churn and pre-release bugs must be conducted before any claims about their relationship to defects can be made. Since there are a large number of software metrics for which the relationship to defects has been studied [27]. Since it is not possible to study all the metrics in this paper, we chose to start with four of them. We have tried to provide all the details of our study in as clear and concise a manner as possible, so that it can be replicated easily on other projects or just other metrics too.

We used one specific approach to determine the thresholds. Other thresholds may not yield exactly the same results, but we found that simply picking the largest or the most complex files yields very similar results. We, however, chose a technique that is empirically based, and takes into account the statistical distribution properties of the file metrics. We derive the thresholds from 1000s of OSS and 205 industrial projects.

Internal Validity: The metrics were extracted from the projects using Understand [2], and Sonar [1]. Any error in these tools can greatly affect the values of the thresholds. However both tools are standard tools and various studies have been carried out using these tools. Furthermore, even if the exact thresholds are slightly changed, the main findings that the top-risk group is the most fault-prone and that it has the lowest defect density, are unlikely to change.

In addition to errors in tools, the difference in tools is also a threat to our results. The difference in tools may affect the values of metrics and thresholds. The difference in tools happened only between OSS and industrial projects (i.e., we used Understand to OSS projects and Sonar to industrial projects). Hence, we believe that we obtained consistent results during each analysis (OSS and industrial projects). The larger problem here is the difference in organizational culture between OSS and industrial projects.

The defect data collection technique is at a commit level and therefore might have both false positives and false negatives. Also, industrial projects consider only customer-reported defects, while OSS projects also include alpha/beta testing, and development defects as it was not practical to separate defects reported only on stable releases for the studied OSS projects. However, such a commit based approach has been the standard in defect prediction literature [7, 19, 27, 30].

Construct Validity: All our results rely on the assumption that the 70%, 80% and 90% percentiles of the overall code are good enough for deriving metrics (which is reported by Alves *et al.* [3]). However, we find that our results as far as metrics are concerned, are not that different from what Alves *et al.* [3] reported.

In Section IV-C, we chose 1,000 projects that had the most similar Median LOC/File to the OSS projects for which we had defect data, in order to derive the thresholds. The choice of representative projects based on Java language and the median of the distribution of project file sizes may have affected our results. It may be that by picking similar projects based on other quantiles or based on the domain, size or other characteristics, would have changed the results. Hence the results should be interpreted within the context: Java projects whose risk thresholds are derived from other Java projects of similar median file size. Further research is needed to determine if the risk thresholds have a consistent relationship to defect density for other languages or for a specific subset of Java projects. We used the median LOC/File to identify the set of projects that could be used to derive the thresholds. Because we removed the smallest projects from our dataset (containing fewer than 200 files), we believe that choice to be a reasonable compromise between the sample size and project similarity.

VIII. Conclusion

In summary we derived the thresholds for three OSS and four industrial projects and assigned the files having a particular value for a metric to a particular risk category depending on the threshold values. Using the defect data for these projects we examined the relationship between the file metrics in each risk category and the defect proneness and defect density of the files. We found that typically defect proneness values monotonically increase with the metric values. However, defect density values typically demonstrate a trend that in some cases appears to contradict the "Goldilocks Conjecture", and in some other cases just has low values only for the files in the very high risk categories for size and complexity metrics. However, we also found some instances when files with small size and complexity metrics had low defect densities. Therefore, although we found some support for findings in recent literature [20, 21, 33] that smaller files have higher defects density, we found further evidence that very large or complex files have lower defect densities and in some cases even lower defect proneness.

In summary, a) we found support for the approach to identify defect-prone files via size thresholds. The complexity thresholds, however, did not work as well, b) we did not find support for approaches that would reduce the size of the largest files by moving that code into other smaller or less complex files. On the contrary, such approaches may increase the risk, and therefore be counterproductive. Hence in conclusion, we found no support for the idea that files with higher defect densities can be identified via thresholds of basic size and complexity metrics. Therefore, any software practitioner who choses to use such thresholds must proceed with caution. We believe that as a result of our study, we as software engineering researchers, need to examine the underlying reasons behind the relationship between existing metrics and software defects, and need to explore other metrics that could be more consistent with respect to defect densities in files.

IX. Acknowledgements

This research was partially supported by JSPS KAKENHI Grant Numbers 15H05306.

References

[1] Sonar, http://www.sonarsource.org/, 2015.

[2] Understand, http://www.scitools.com/, 2015.

[3] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*, pages 1–10, 2010.

[4] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation. *ACM Commun.*, 27(1):42–52, 1984.

[5] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Looking for bugs in all the right places. In *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72, 2006.

[6] S. Benlarbi, K. E. Emam, N. Goel, and S. Rai. Thresholds for object-oriented measures. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE)*, pages 24–38, 2000.

[7] T.-H. Chen, S. Thomas, M. Nagappan, and A. Hassan. Explaining software defects using topic models. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 189 –198, 2012.

[8] D. Coleman, B. Lowther, and P. Oman. The application of software maintainability models in industrial software systems. *J. Syst. Softw.*, 29(1):3–16, 1995.

[9] B. T. Compton and C. Withrow. Prediction and control of ada software defects. *J. Syst. Softw.*, 12(3):199–207, 1990.

[10] K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai. The optimal class size for object-oriented software. *IEEE Trans. Softw. Eng.*, 28(5):494–509, 2002.

[11] K. Erni and C. Lewerentz. Applying design-metrics to object-oriented frameworks. In *Proc. Int'l Symposium on Software Metrics (METRICS)*, pages 25–26, 1996.

[12] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, Sept. 1999.

[13] M. Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.

[14] V. French. Establishing software metric thresholds. In *Proc. Int'l Workshop on Software Measurement (IWSM)*, 1999.

[15] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[16] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, Nov. 2012.

[17] L. Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, 1997.

[18] L. Hatton. Does OO sync with how we think? *IEEE Software*, 15(3):46–54, 1998.

[19] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort aware models. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*, pages 1–10, 2010.

[20] A. Koru, D. Zhang, K. El Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Trans. Softw. Eng.*, 35(2):293–304, 2009.

[21] G. Koru, H. Liu, D. Zhang, and K. E. Emam. Testing the theory of relative defect proneness for closed-source software. *Empirical Softw. Eng.*, pages 577–598, 2010.

[22] S. Matsumoto, Y. Kamei, A. Monden, and K. Matsumoto. Comparison of outlier detection methods in fault-proneness models. In *Proc. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 461–463, 2007.

[23] T. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, SE-2(4):308 – 320, 1976.

[24] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 11–20, 2009.

[25] B. A. Nejmeh. Npath: a measure of execution path complexity and its applications. *ACM Commun.*, 31(2):188–200, 1988.

[26] R. Shatnawi, W. Li, J. Swain, and T. Newman. Finding software metrics threshold values using ROC curves. *J. Soft. Maint. Evol.*, 22(1):1–16, 2010.

[27] E. Shihab. An exploration of challenges limiting pragmatic software defect prediction. *PhD Thesis, School of Computing, Queen's University, Kingston, Ontario, Canada*, 2012.

[28] D. I. Sjøberg, B. Anda, and A. Mockus. Questioning software maintenance metrics: a comparative case study. In *Proc. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 107–110, 2012.

[29] D. I. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Trans. Softw. Eng.*, 99(PrePrints):1, 2013.

[30] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Conf. on Mining Software Repositories (MSR)*, pages 1–5, 2005.

[31] D. Spinellis. A tale of four kernels. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 381–390, 2008.

[32] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *IEEE Int'l Conf. on Software Maintenance (ICSM)*, pages 179 –188, 2009.

[33] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Eng.*, 13(5):539–559, 2008.

[34] 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 *Proc. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 443–445, 2007.