# The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes

di Biase, Marco; Rastogi, Ayushi; Bruntink, Magiel; van Deursen, Arie

**DOI**
https://doi.org/10.5281/zenodo.2606632

**Publication date**
2019

**Document Version**
Accepted author manuscript

**Published in**
TechDebt 2019 - International Conference on Technical Debt

# The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes

Marco di Biase*†, Ayushi Rastogi†, Magiel Bruntink*, Arie van Deursen†
*Software Improvement Group - Amsterdam, The Netherlands †Delft University of Technology - Delft, The Netherlands
Email: *[m.dibiase, m.bruntink]@sig.eu, †[a.rastogi, Arie.vanDeursen]@tudelft.nl

***Abstract*—Existing maintainability models are used to identify technical debt of software systems. Targeting entire codebases, such models lack the ability to determine shortcomings of smaller, fine-grained changes. This paper proposes a new maintainability model – the Delta Maintainability Model (DMM) – to measure fine-grained code changes, such as commits, by adapting and extending the SIG Maintainability Model. DMM categorizes changed lines of code into low and high risk, and then uses the proportion of low risk change to calculate a delta score. The goal of the DMM is twofold: first, producing meaningful and actionable scores; second, compare and rank the maintainability of fine-grained modifications.**

**We report on an initial study of the model, with the goal of understanding if the adapted measurements from the SIG Maintainability Model suit the fine-grained scope of the DMM. In a manual inspection process for 100 commits, 67 cases matched the expert judgment. Furthermore, we report an exploratory empirical study on a data set of DMM scores on 3,017 issue-fixing commits of four open source and four closed source systems. Results show that the scores of DMM can be used to compare and rank commits, providing developers with a means to do root cause analysis on activities that impacted maintainability and, thus, address technical debt at a finer granularity.**

## I. Introduction

Software maintainability defines the ease with which software can be modified, *e.g.*, to correct defects, or to improve its functionality [1]. Measuring and improving maintainability helps managing technical debt, a concept introduced by Cunningham [2] and then refined by Fowler [3], [4]. To grasp technical debt, Ernst *et al.* [5] noted that code analysis and measurements are the main concrete methods used to understand technical debt.

Maintainability is a complex concept, and past research shows efforts towards data-driven approaches to indicate maintainability, with several models proposed [6]. Some of these models are in active use in research or in industry, such as the SQALE method [7], the Software Improvement Group Maintainability Model (SIG-MM) by Heitlager *et al.* [8], or the QUAMOCO model [9]. These models take the source code of complete software systems at fixed moments in time (*i.e.*, snapshots) as their primary units of maintainability analysis. Such an analysis matches well with the relatively low update and release frequencies of larger (legacy) software systems.

In recent years however, development is driven by an increasing level of change granularity. Smaller changes are being implemented in software releases [10], continuous integration tools ensure that every committed code change is integrated into the primary line of development [11], and developers use fine-grained mechanisms such as pull requests to review and accept code changes [12]. To support these modern development practices, maintainability measurement models need to take fine-grained code changes (*i.e.*, a single, or limited number of commits) as their primary unit of analysis. In this scenario, developers could address bad code that creates technical debt at a finer level of abstraction, and promptly use actionable suggestions rather than refactor larger portions of code. In fact, nonfunctional requirements such as maintainability are a concern during the modern development processes because they contribute to the completion of features [13] and managing technical debt repercussions drain project resources [5].

Despite measuring differences in code metrics (*i.e.*, churn or McCabe Complexity [14]) provide a superficial insight about fine-grained change-maintainability, they lack the breadth of formalized models available for complete software systems. Such models, though, are not suitable to measure fine-grained code maintainability as they carry the burden of the full system codebase. In our view, a fine-grained maintainability model should rely on code measurements able to detect the amount of risk introduced over the total change performed, meaningfully report the maintainability of fine-grained code changes abstracting from the full system codebase, and use a scoring system that allows result comparison aside from the system it is applied.

In this paper we propose a new model for this purpose, which we named the Delta Maintainability Model (DMM). The DMM measures the maintainability of a code change as ratio between low risk code and the overall code modified. The DMM identifies code riskiness by reusing software metrics and risk profiles of the SIG-MM, while applying new aggregations and scoring for software metric deltas at the level of fine-grained code changes like commits or pull requests, instead of aggregating at the system level. The DMM does not aim to replace the SIG-MM, rather, complement its system-level analysis with direct support for fine-grained code changes.

In the study, we first focus on understanding if the SIG-MM measurements suit the fine-grained scope of the DMM. In a manual inspection process, we measure a sample of 100 code changes against the intuition and expertise of the main authors. Then, we analyze data of the DMM's outcomes on 3,017 fine-grained code changes, originating from four open-source and four closed-source systems, to understand whether the score produced suits the needs of ranking and interpret-

ing the maintainability of code changes. We summarize the contributions of this paper as follows:

- The proposal and initial assessment of the Delta Maintainability Model (DMM), a novel approach to measure the maintainability delta of fine-grained code changes. DMM is an extension of, and a complement to the SIG Maintainability Model.
- A data set of 2,336 fine-grained open-source code changes with the initial DMM Scores to enable further research in this direction. These code changes are a subset of our data set, of which the closed-source code changes cannot be disclosed.

## II. Related Work

Aggarwal *et al.* [15] proposed a model to measure software maintainability based on code readability, documentation quality, and understandability of software. They transform measures into fuzzy values, using domain experts to process results and quantify maintainability of a system.

Antonellis *et al.* [16] proposed a way to map object-oriented metrics presented by Chidamber *et al.* [17] to the characteristics of the ISO 9126 model to evaluate a software system's maintainability. They applied the methodology to an Open-Source system, demonstrating that software maintainability was measurable with a systematic process.

Heitlager *et al.* [8] proposed the SIG-MM to measure maintainability for software systems, which we extensively present in Section III. Later studies on SIG-MM provided approaches with which the model measurements could be aggregated to allow a comparison between systems [18]–[20].

The work of Bakota *et al.* [21] uses a probabilistic approach to compute high-level quality features that uses expert knowledge as well. It integrates the uncertainty from the lack of consensus, and the maintainability value is a probability distribution.

The QUAMOCO model has been proposed by Wagner *et al.* [9] to build an approach that integrates abstract quality attributes and concrete quality assessments. Such a model provides an assessment methodology that integrates with their meta-model definition, and summarizes how to conduct a quality check for different kinds of software.

The SQALE method [7] proposed by Letouzey and implemented in practice by SonarQube [22] relies on so-called Indices to rate various aspects of code quality. Recently, though, SonarQube shifted towards continuous inspections of code that integrates with DevOps toolchain.

The work by Conradt *et al.* [23] provided a first framework that adapts to the needs of incremental quality and maintainability checks on code. This work resulted in Teamscale [24], [25], that measures multiple aspects of code such as file size, method length, clones, *etc.* Teamscale can quickly provide metrics and compute change in measurements without doing a system-wide measurement, providing real-time feedback to developers. Furthermore, Teamscale remembers which commits caused the change in metric, thus allowing root cause analysis. Teamscale reports finding *"resulting from a violation of a metric threshold (such as a file which is too long) or revealed by a specific analysis such as clone detection or bug pattern search"* [24]. Such violations are counted against a commit and used to account for quality. Teamscale, though, is not able to rank the maintainability of a single code change (or any aggregation) and combine the results in a quality or maintainability score.

## III. Background: SIG Maintainability Model

The maintainability model that we use as starting point is the one proposed by Heitlager *et al.* (SIG-MM in the paper) [8]. It has been used in formal assessments of maintainability in an industrial context for billions of lines of code, thousands of projects, and a wide variety of application domains and programming languages and technologies. SIG-MM was designed to conduct system-level assessments with technology independence, ease of understanding, root cause analysis, and actionability as design choices. The model was originally proposed in 2007, with later refinements [18]–[20], [26].

A common point of departure for maintainability models, including SIG-MM, is the ISO standard 25010 for software product quality [27]. This standard identifies maintainability characteristics such as analyzability, modifiability, testability, modularity, and reusability. What SIG-MM does is map these ISO maintainability characteristics onto source code metrics, as shown in Table I. The metrics relevant to this paper are defined in Table II, discarding Volume, Component Balance and Independence as unlikely being impacted by a fine-grained change.

The SIG-MM takes absolute measurement numbers and turns these into a *rating* in comparison to a benchmark of hundreds of measured industrial systems. This rating is a (real) number between 0 and 5, or, colloquially, a way to assign between one and five stars to the maintainability of a system.

The process to move from measurement to rating is depicted in Figure 1. Central to this process are *risk profiles* [18]. A risk profile groups metric outcomes into (risk) bins, labeled low, medium, high and very-high. The threshold values defining each bin are obtained by a statistical approach proposed by Alves *et al.* [18]. The risk profile then reflects the *percentage of code* in the low to very high-risk categories. Risk profiles can be created for individual metrics at the unit level, and aggregated all the way up to the system level.

Once the risk profile of a system is known, it is turned into a rating, using the relative size for the moderate, high and very-high risk profiles for each system property. The rating is again based on thresholds, this time expressing that, *e.g.*, a system in which at most 3.3% of its Volume is in the high-risk category (five stars in Figure 1) can be considered substantially easier to maintain than when, say 15% of its Volume is in the high risk category (two stars only). The statistical procedure for computing the rating thresholds is described by Alves *et al.* [19].

## IV. Context and Challenges

While traditional maintainability models such as SIG-MM focus on system-level maintainability, our focus is on assessing

TABLE I. MAPPING ISO MAINTAINABILITY SUB-CHARACTERISTICS TO SYSTEM PROPERTIES

| Sub-characteristic | Volume | Duplication | Unit Size | Unit Complexity | Unit Interfacing | Module Coupling | Component Balance | Component Independence |
|---|---|---|---|---|---|---|---|---|
| Analysability | o | o | o | | | | o | |
| Modifiability | | o | | o | | o | | |
| Testability | o | | | o | | | | o |
| Modularity | | | | | | o | | o |
| Reusability | | | o | | o | | o | |

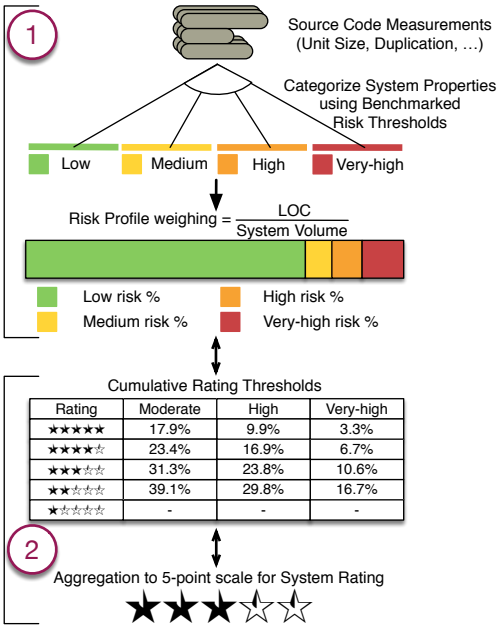



FIGURE 1. SIG-MM APPROACH TO MOVE FROM SOURCE CODE MEASUREMENTS TO SYSTEM-WIDE RATINGS

the quality of actual changes. We seek an assessment of a fine-grained change (commit) independent of the quality of the surrounding bigger system, given that developers deal with a high number of modifications to such systems on a daily basis.

As for system level maintainability models, our aim is to provide an assessment that is technology independent, and that supports understanding, root cause analysis, and actionability (in line with [8]). Creating such an assessment method requires addressing two challenges: measuring at the commit level, and scoring and comparing commits.

### A. Commits as the primary unit of measurement

The core issue with current maintainability measurement models is that their primary unit of analysis is a snapshot of a system's (entire) code base. With the SIG-MM, for instance, this leads to small changes to large systems typically having a very minor impact on the maintainability rating. The reason is that changes are always weighted towards the entire system's code base; this was in fact a design goal of the SIG-MM.

Consider for instance issue #402331 (a bug) in Mozilla Rhino,[1] fixed by one commit that changed about 200 lines of code.[2] The impact on the SIG-MM maintainability rating is just $-0.007$, a negligible delta on a scale of -5 to 5 due to the weighting against Rhino's entire code base.[3] This result does not suggest any change in maintainability. In practice, however, this change introduced less maintainable code, which we discuss later in details. This delta is also only meaningful in relation to Rhino's code base prior to the change. A similar change applied to a far smaller system would have a far greater variation in maintainability.

The challenge here is to define a model that no longer weights against a system's entire code base, but instead introduces a meaningful, actionable approach to measure commits in their own right, irrespective of the parts of the code base that are not involved in the changes.

### B. Scoring and comparison of commits

With a new unit of measurement, *i.e.*, commits, also comes a new challenge to construct a new approach to score and compare. Similar to SIG-MM's context for systems, a specific need in our circumstances is to score and compare commits to each other. Ideally, this would mirror the ability to rank and benchmark commits, analogously to the approach by Baggen *et al.* to rank systems [20].

An issue to consider here is that many software (maintainability) metrics follow heavy-tailed distributions [28]–[30], which calls for additional steps to normalize values to facilitate meaningful comparison. The challenge here thus consists of either creating scoring mechanisms that do not require additional normalization steps, or to define normalization approaches similar the proposal by Alves *et al.* [19].

## V. THE DELTA MAINTAINABILITY MODEL

The general idea behind the DMM is to calculate the proportion of a change that is beneficial for maintainability, given the various code properties already provided by the SIG-MM. A change is considered to consist of LOCs being added and removed (similar to a 'diff') to units and modules touched by the change. Adding LOCs to units or modules that stay low risk (see Table II) is considered a beneficial change, as well as removing LOCs from high risk units or modules. However, adding LOCs to high risk, or removing LOCs from low risk units or modules, is then considered harmful to maintainability.

The DMM utilizes the SIG-MM as base for its workings because it uses a straightforward way of abstracting metrics to maintainability properties. Code measurements are summarized in risk profiles according to threshold per each system property and threshold are defined according to a set of systems used as benchmark. Furthermore, this approach allows to measure code written in different programming languages by generalizing measurements.

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=402331
[2] https://github.com/mozilla/rhino/commit/262602
[3] The absolute maintainability ratings are respectively 1.9487 for the system before the bug fix was applied, and 1.9414 after the bug fix.

TABLE II. DESCRIPTIONS OF THE SIG-MM SYSTEM PROPERTIES AND THEIR THRESHOLDS FOR QUALIFYING CODE AS LOW RISK.

| System Property | Description | Low risk code criteria |
|---|---|---|
| **Duplication** | The degree of (textual) duplication in the source code of the software product. A line of code is considered redundant if it is part of a code fragment (larger than 6 lines of code) repeated literally (modulo white-space) in at least one other location in the source code. | All non-duplicated code. |
| **Unit Size** | Size of the source code units, based on Lines Of Code (LOC). Size is determined from the number of lines of code (excluding lines consisting of only white space or comments). | Units with at most 15 LOC. |
| **Unit Complexity** | The degree of complexity in the units of the source code. The notion of unit corresponds to the smallest executable parts of source code, such as methods or functions. Complexity is measured using McCabe's cyclomatic complexity [14]. | Units with at most 5 McCabe complexity. |
| **Unit Interfacing** | The size of the interfaces of the units in terms of the number of interface parameter declarations (formal parameters). | Units with at most 2 parameters. |
| **Module Coupling** | The coupling between modules, measured by the number of incoming dependencies. The notion of module corresponds to a grouping of related units, typically a file. | Modules with at most 10 fan-in. |

Table II contains the system properties utilized by the DMM brought over from the SIG-MM, with a short description and the relative value that define their low-risk threshold.

In the following, we first provide the definition of the DMM and then present a calculation example. Figure 2 provides an overview of the model and its underlying calculations.

### A. Model Calculation

The calculation of the DMM consists of two levels. The first level in Figure 2a describes how a code change maps into a Risk Profile - a concept originally from SIG-MM but adapted for small code changes. The second level in Figure 2b combines all Risk Profiles for a code change to generate a DMM score. Below we provide simple instructions to generate DMM scores from two inputs files **(1)** changed within a commit (Fn) and the original files (or parent files) prior to the change (Fn'). Formal definitions of the model are available in our technical report [31]. Per each file:

**(2)** the DMM measures and classifies into a Risk Profile (low, med, high, very-high) the five Code Properties listed in Table II.

**(3)** Measurements are taken in lines of code according to the specifications of the SIG-MM [19].

**(4)** Next, the model computes the various Deltas, *i.e.*, the difference measured in lines of code before and after the change performed on a file. Deltas for a Code Property are computed for all Risk Profile. When the difference of the Delta is negative, we define it a Delta Decrease (Increase otherwise) and transform its value to absolute number.

To aggregate Deltas at commit level, following Figure 2b and considering step **(4)** from Figure 2a as input:

**(5)** the DMM sum all the Delta values per each Code Property, resulting in each commit having two values per Risk Profile: one for the Increases, one for the Decreases.

**(6)** Notably, increases in LOC in low risk profile is a good, highly-maintainable change, as well as the decrease in LOC in medium, high and very-high category. These values account for what we define Low Risk Profile Delta. On the other hand, decrease in LOC in low risk category and increase in LOC in medium, high and very-high category are deemed as low-maintainability changes and account for High Risk Profile Delta.

**(7)** Finally, the Delta Score for each Code Property is the fraction of highly maintainable Low Risk Profile Delta out of the total.

**(8)** When the five Code Properties are aggregated (here using mean) we refer to it as the Delta Maintainability Score (DMM Score).

Both the Delta Score and the DMM Score range between 0 and 1, to be interpreted as 0 being the lowest and 1 the highest maintainability. Threats connected to aggregation techniques for various DMM choices are discussed in Section VI-C.

### B. Calculation example

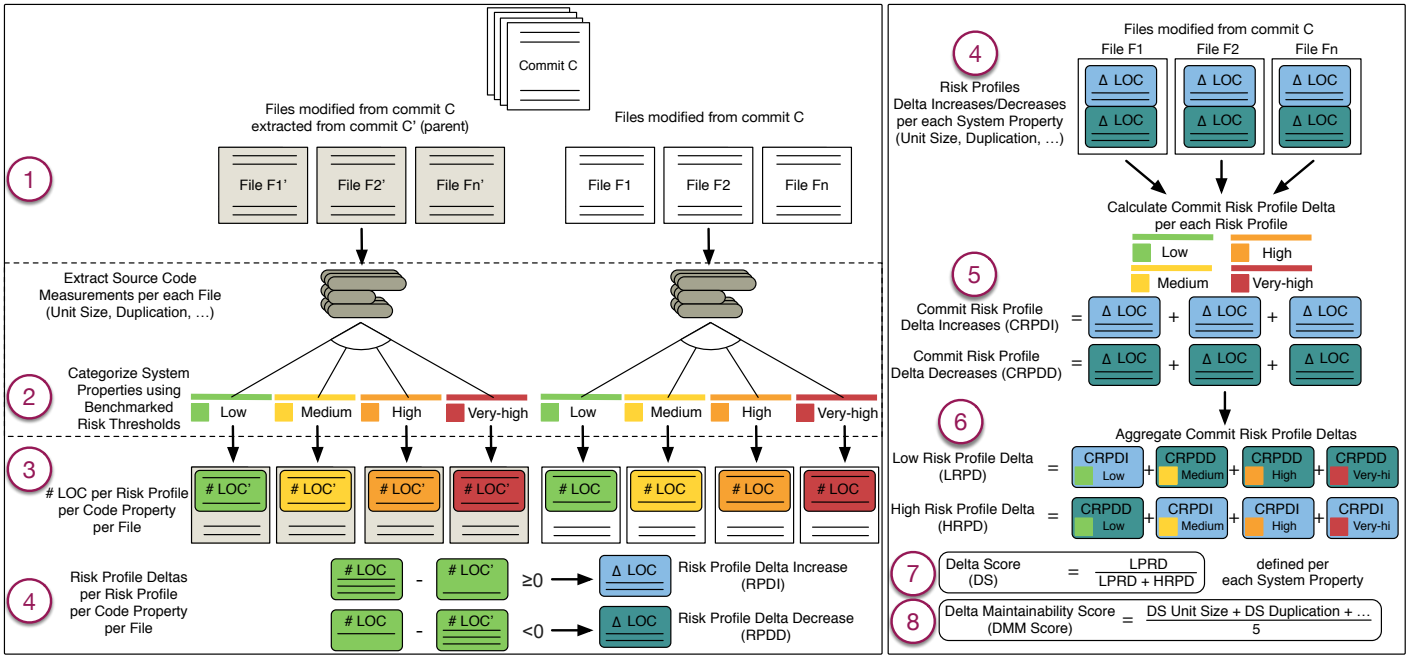
We present a walkthrough example to explain the approach used to obtain metric deltas from measurements and then obtain the DMM Scores. Table III contains the data for this example. For simplicity we illustrate only the Delta Unit Size, as the rationale for the other system properties is identical. In fact, the DMM maps its argument to LOC for all Risk Categories and Code Properties as specified by the SIG-MM [19].

We use issue #402331[4] for Mozilla Rhino, already discussed in Section IV. Mozilla Rhino is a system written in Java comprising approximately 244K lines of code.

The general steps for moving from source code measurements to Risk Profile Delta are showcased in Figure 2a. The figure also refers to the steps 1-4 in Section V-A. For our example, the steps taken involve: **(1)** Issue #402331 is fixed by commit id 262602.[5] It changes files `Codegen.java` and `OptRuntime.java`. These files are analyzed from the system checked out at commit id 833a2a as well (parent of 262602); **(2)** measure all Code Properties included in the DMM using the SIG-MM Benchmarked Risk Profile Thresholds; **(3)** file measurements for Unit Size are shown in Table IIIa; **(4)** measure the Risk Profile Delta for Unit Size as shown in Table IIIb. The Risk Profile Delta still has positive or negative values. Table IIIc shows the respective values for

[4]https://bugzilla.mozilla.org/show_bug.cgi?id=402331

[5]https://github.com/mozilla/rhino/commit/262602




(A) LEVEL 1: RISK PROFILE DELTAS. (B) LEVEL 2: DELTA SCORES.

FIGURE 2. OVERVIEW OF THE DMM.

Risk Profile Delta Increases and Decreases, that removes the sign from the Risk Profile Delta.

Having completed the steps from Figure 2a, the collected increases and decreases are converted into a single score. The general approach is depicted in Figure 2b, and consists of the following steps for our example: **(5)** aggregate the Risk Profile Delta Increases/Decreases at commit level by sum of each measurement; for the walkthrough example, Table IIId shows the aggregation for Unit Size; **(6)** compute the Low/High Risk Profile Delta by aggregating measures. Negative Deltas are aggregated with their absolute value, resulting in: LRPD $= 4 + 0 + 0 + 0 = 4$; HRPD $= 25 + 0 + 0 + 13 = 38$; **(7)** the Delta Score Unit Size is: $\frac{4}{4+38} = \frac{4}{42} \approx 0.09$; **(8)** for issue #402331, Mozilla Rhino, the Delta Scores for the other system properties are: Unit Complexity $= 0.09$, Unit Interfacing $= 0.327$, Module Coupling $= 0.391$, Duplication $= 0.795$. Therefore, the computed value for the DMM Score is: $\frac{0.09+0.09+0.327+0.391+0.795}{5} \approx 0.341$.

## VI. STUDY DESIGN

The proposed DMM has been subject to an initial assessment study, consisting of two parts. First, the first two authors of this paper performed a manual effort to assess the design and implementation of DMM. In particular, we want to test whether the SIG-MM metric are still valid and extend to the fine-grained scope of analysis of the DMM. Second, a more general exploratory study was done on the DMM scores for a large number of code changes for several open and closed source systems.

The first part of the study had the aim to answer the following research question:

TABLE III. CALCULATION EXAMPLE - UNIT SIZE - ISSUE #402331 MOZILLA RHINO

| commit id | File name | Low | Medium | High | Very-High |
|---|---|---|---|---|---|
| 262602 | Codegen.java | 359 | 465 | 1138 | 1935 |
| | OptRuntime.java | 147 | 0 | 0 | 0 |
| 833a2a | Codegen.java | 355 | 465 | 1138 | 1922 |
| | OptRuntime.java | 172 | 0 | 0 | 0 |

(A) LOC RISK PROFILE PER FILE IN COMMIT 262602 AND 833A2A

| File name | Low | Medium | High | Very-High |
|---|---|---|---|---|
| Codegen.java | 4 | 0 | 0 | 13 |
| OptRuntime.java | -25 | 0 | 0 | 0 |

(B) RISK PROFILE DELTA PER FILE

| | Increases | | | | Decreases | | | |
|---|---|---|---|---|---|---|---|---|
| File name | Low | Medium | High | Very-High | Low | Medium | High | Very-High |
| Codegen.java | 4 | 0 | 0 | 13 | 0 | 0 | 0 | 0 |
| OptRuntime.java | 0 | 0 | 0 | 0 | 25 | 0 | 0 | 0 |

(C) RISK PROFILE DELTA INCREASES/DECREASES PER FILE

| | Increases | | | | Decreases | | | |
|---|---|---|---|---|---|---|---|---|
| Commit id | Low | Medium | High | Very-High | Low | Medium | High | Very-High |
| 262602 | 4 | 0 | 0 | 13 | 25 | 0 | 0 | 0 |

(D) COMMIT RISK PROFILE DELTA INCREASES/DECREASES

**RQ1.** To what extent do the code metrics defined by SIG-MM remain meaningful when applied to fine-grained code changes (by DMM)?

After this step, the exploratory empirical study focused primarily on understanding whether the DMM scores represents a viable proposition to measure and compare the maintainability of fine-grained changes. The following research question was posed for this purpose:

**RQ2.** To what extent can the DMM Scores be used to rank and compare the maintainability of fine-grained code changes?

### A. Data set creation

To answer both our research questions, we created a historical data set of issue fixing commits (*e.g.*, commits that fix a bug) performed on several open- and closed source software systems.

*1) Subject systems:* We included four large open source systems belonging to the dataset of Herzig *et al.* [32], which consists of manually classified issues.[6] We used this dataset as being widely used in the past by multiple studies, and because the projects feature an easy-to-explore issues tracker and version control system (VCS), which we require to relate issues to the code changes that resolved them.

Furthermore, we included four closed source systems that the authors have access to under non-disclosure conditions. For these systems only general information can be provided. Cs1 is a tool written in Java to do static analysis on source code; cs2 and cs3 are respectively the back-end and the front-end for a web application written in Groovy/GSP. Cs4 is a web application written in C#/ASP.NET. Table IV reports the characteristics of the systems included in the data set.

TABLE IV. CHARACTERISTICS OF SYSTEMS IN THE DATA SET.

| System | Language | # Issues | KLOC |
|---|---|---|---|
| Apache Jackrabbit | Java | 1532 | 338 |
| Apache Tomcat | Java | 294 | 308 |
| Apache httpcomponents-client | Java | 154 | 67 |
| Mozilla Rhino | Java | 356 | 244 |
| Closed source 1 (cs1) | Java | 488 | 240 |
| Closed source 2 (cs2) | Groovy | 92 | 9.6 |
| Closed source 3 (cs3) | Groovy/GSP | 36 | 3.2 |
| Closed source 4 (cs4) | C#/ASP.NET | 65 | 168 |
| Total | | 3,017 | 1,378 |

*2) Extraction of commits that resolve issues:* Two approaches were applied to obtain the commit(s) that resolve an issue, depending on whether they are open- or closed source.

*a) Open source projects:* We started from the list of issues provided by Herzig *et al.*, selecting those classified as *BUG*, *Request For Enhancement* (RFE, adaptive maintenance), and *IMPROVEMENT* (perfective maintenance) [32, Table II]). We excluded the remaining issues like those that fix documentation, tests, *etc.* To extract the issue fixing commits, we analyzed the VCS log of each system and identifying the exact match of the issue identifier in commit messages. When an issue was fixed by more than one commit, we included all commits. This process allowed us to identify fixing commits for 2,336 issues in the original data set.

*b) Closed source projects:* For the closed source projects a more limited amount of information was available to the authors, constraining the analysis to just bug fixes.

To identify bugs we used the same strategy that Herzig *et al.* used in their work [32]. The first author inspected the VCS logs of the projects, selecting commits that were made to fix bugs, as well as inspected manually their change set using *git diff*. When issues were resolved by multiple commits, we read the messages of the following commits after the first. Commits were included in our dataset if the context was the same (*i.e.*, changing the same file(s) of the previous commit in relevant parts). For project *cs1* we had access to its Jira Issue Tracker, that we used to first select issues that could be classified as bugs. To assess the accuracy of the classification, we manually analyzed a representative sample of 20 bug fixes using the same technique described earlier, finding no misclassified issues.

Finally, our data set consisted of 3,017 issues with a matched issue fixing commit.

### B. Manual assessment of DMM scores

To answer RQ1, we sampled 100 issues from the data set constructed earlier and performed a manual assessment of the issue-fixing commits and their DMM scores.

*1) Sampling:* The sampling was done with the following criteria: (1) comprise all the open-source projects in the dataset; (2) equal number of issues per project (25 issues per project); (3) all issues are fixed by a single commit; (3) 50 issues have DMM Score $< 0.5$, while the remaining 50 have DMM Score $\geq 0.5$. The sample was constructed such that it represents equally the four open-source projects and their maintainability. Therefore, we drew 25 randomly sampled issues per project. For simplicity, we selected only issues fixed by a single commit. The model, however, works identically for any number of commits.

*2) Assessment:* The first two authors were provided with a document listing 100 URLs linking to code diffs (on GitHub) of the sampled issues. The instruction given to both authors was to read each diff and assess the maintainability risk of the change. They were asked to categorize the change into either Good or Bad for maintainability, keeping in mind the code properties of the SIG-MM included in DMM (see Table II).

The author judgements were done in two rounds. In the first round, the first author inspected all 100 issue fixes in the sample, while the second author inspected a randomly selected set of 50 issues. In the second round, the authors worked together to achieve consensus on mismatching scores using the negotiated agreement technique [33]. We discuss possible bias in section VI-C.

*3) Comparing author judgements and DMM scores:* Finally, the authors compared their judgements with the DMM

[6] https://www.st.cs.uni-saarland.de/softevo/bugclassify

scores, for both the individual code properties and the overall score. To ease comparison, the DMM scores were first simplified into either a Good or Bad category as follows:

- $\geq 0.5$ means Good for maintainability,
- $< 0.5$ means Bad for maintainability.

The threshold score of 0.5 was chosen since it reflects the point at which a change contains proportionally more low than high risk changes (step 7, Figure 2). We would like to stress the experimental nature of this threshold, provided that we have no empirical data to support this value choice. We evaluate and discuss this choice in the result section for RQ1.

### C. Threats to validity

*a) Construct validity:* We identify limits of our study regarding the definition of maintainability that applies to small, fine-grained changes. Provided that no previous study has tried to formalize such concept, we try to bring one first definition.

Moreover, the DMM assumes certain aggregation in risk profiles for the measurements considered, with the methodology for deriving their thresholds proposed in the past and widely benchmarked [18].

The DMM definitions might be sensitive to aggregation techniques such as summation and mean. Previous studies showed that aggregations could impact defect prediction models [34]. Additionally, measuring the maintainability of a software system (or change) as defined by the DMM, some Code Properties might have more importance than others. Unfortunately, lack of empirical data prevents us addressing the two aforementioned possibilities. Future research should provide evidence about impact of aggregation techniques to metric-based maintainability assessment and relative metric relevance being better at estimating maintainability.

*b) Reliability:* Despite our dataset sampled to answer our first research question does not contain issues fixed by multiple commits, our data closely matches that of an average pull request. Our study sample considers code changes that have median churn of 13.5. Gousios *et al.* [12] shows that the median number of commits per pull request is 1, and the median churn per pull requests is 20.

The manual analysis performed poses an inherent threat to the assessment of the DMM. Ideally, experts of particular systems are the best candidates as researchers might not be familiar with a system's specifics. We mitigate this by gathering a deep understanding of the code analyzed, by repeated inspections, and by employing the negotiated agreement technique [33].

*c) Conclusion validity:* Threats to conclusion validity can be identified in the way our study evaluates the model. Data selected for the systems (Section VI-A1) are retrieved from a known dataset. This, although represents a reliable source, still has breadth limitations. Finally, we filter their VCS using only the master branch, to make sure that only merged commits were selected as subject for our study.

## VII. Results and Analysis

*RQ1: To what extent do the code metrics defined by SIG-MM remain meaningful when applied to fine-grained code changes (by DMM)?*

TABLE V. RESULTS OF THE MANUAL ASSESSMENT. A1 AND A2 ARE AUTHORS 1 AND 2. A1 $\neq$ A2 IS WHERE THEY DISAGREED.

| DMM Score | A1+A2 Good | A1+A2 Bad | A1 $\neq$ A2 |
|---|---|---|---|
| Good | **14** | 12 | 1 |
| Bad | 6 | **16** | 1 |

| DMM Score | A1 Good | A1 Bad | |
|---|---|---|---|
| Good | **16** | 5 | - |
| Bad | 7 | **21** | - |

Table V shows results from the manual assessment performed by the two authors. They manually classified 48 out of 50 issues assigned to them as *good* or *bad* in terms of maintainability. Out of the 48 issues classified (see the first confusion matrix in Table V), the result of the two authors matched for 30 issues (14 good and 16 bad). The two authors, then, discussed each of the remaining 18 issues until a consensus was reached. Finally, we manually assessed the maintainability for 48 issues. These manually assessed and cross-validated issues where then matched against their actual DMM scores. Out of the 48 issues classified, manual classification matched DMM scores for 30 issues.

Likewise, the first author classified 50 other issues manually. This time, 49 out the 50 issues were classified. Out of the 49 issues classified (see the second confusion matrix in Table V), the classification matched with the DMM scores in 37 cases (16 good and 21 bad). In total, 67 issues were assigned the correct score by the DMM out of the 97 issues that were manually classified, corresponding to an accuracy of 69%.

Further, we analyzed the issues we could not manually classify or were incorrectly ranked by DMM. The three issues which the authors could not manually classify are JCR-3183[7] and JCR-2010[8] for Apache Jackrabbit, and issue 48049[9] for Apache Tomcat. The first two issues not classified had two common characteristics: (a) they involved large code changes spanning several files ($>$ 200 lines of code change; $>$ 3 files) (b) files containing both good and bad changes in terms of maintainability. These aspects made hard for the authors to decide. Finally, the third issue renamed one variable which is too small of a change to be classified.

To gather more insight on the manual assessment mismatches, we separate the DMM scores in equal thirds. Selecting 0.33 (low DMM Score) and 0.66 (high DMM Score) as separation values, we expected that maximum misclassification would happen in the range where DMM score is around 0.5 (*i.e.*, DMM $\in (0.33, 0.66)$) as these are cases where separating well and badly maintainable changes gets challenging.

The data analysis confirms our hypothesis, as shown in Table VI: we found that 3 out of 14 issues (14%) with DMM

[7]https://issues.apache.org/jira/browse/JCR-3183
[8]https://issues.apache.org/jira/browse/JCR-2010
[9]https://bz.apache.org/bugzilla/show_bug.cgi?id=48049

TABLE VI. MISMATCHED CLASSIFIED ISSUES VS. DMM SCORE

| DMM Score | # Mismatched | # Data points | # Unclassified |
|---|---|---|---|
| $\leq 0.33$ | 3 | 14 | 0 |
| $\in (0.33, 0.66)$ | 22 | 56 | 1 |
| $\geq 0.66$ | 5 | 30 | 2 |
| All | 30 | 100 | 3 |

score less to or equal than 0.33 were incorrectly classified. In the high DMM score group, 5 out of 30 issues (16%) mismatch the manual assessment. Finally, 39% of the data (22 out of 56 issues) with DMM score between 0.33 and 0.66 is incorrectly classified.

We derived additional insights into the DMM scores from the experiences of the two authors who did the manual assessment. They collectively expressed difficulty in identifying duplicates. Duplication, in DMM, is checked in a file, considering duplicates at least 6 consecutive lines. For very small changes, duplication cannot be defined. For bigger changes, the two authors used some basic methodologies to search code in the files to accelerate and facilitate this (*e.g.*, search in files and the GitHub code search functionality). Nonetheless the GitHub user interface proved to be a non-optimal solution because of the visualization limitations.

Another challenge is when larger changes span different files comprising of both good and bad maintainable changes. In cases of large code changes, deciding on one category is difficult. Over time, this still can be considered a suboptimal change. The assessment for these specific cases asks for a judgement call.

In light of this further analysis on the DMM scores we see that the scores follow the intuition of the manual classification. From this observation we can conclude that the five SIG-MM measures in the DMM suit the end goal of measuring fine-grained maintainability and maintain their meaningfulness within the DMM. We include the list of issues, their manual assessment by the two authors and their DM Score in the additional dataset attached to our Technical Report [31].

*RQ2: To what extent can the DMM Scores be used to rank and compare the maintainability of fine-grained code changes?*

TABLE VII. SUMMARY STATISTICS FOR DMM SCORES.

| | All Data n = 3,017 | | Open Source n = 2,336 | | Closed Source n = 681 | |
|---|---|---|---|---|---|---|
| DM Score | mean | sd | mean | sd | mean | sd |
| Maintainability | 0.65 | 0.24 | 0.65 | 0.23 | 0.68 | 0.28 |
| Unit Size | 0.48 | 0.38 | 0.46 | 0.37 | 0.54 | 0.41 |
| Unit Complexity | 0.57 | 0.39 | 0.54 | 0.38 | 0.65 | 0.40 |
| Unit Interfacing | 0.67 | 0.38 | 0.67 | 0.38 | 0.68 | 0.38 |
| Module Coupling | 0.77 | 0.34 | 0.76 | 0.34 | 0.78 | 0.34 |
| Duplication | 0.78 | 0.34 | 0.78 | 0.34 | 0.77 | 0.35 |

Table VII contains the means and standard deviations for the DMM scores. Given the minor differences in the mean visible between the open- and closed source subsets, in particular for Unit Size and Unit Complexity, and considering standard deviations, we evaluate these differences too small to support further splitting of the data set.

Figure 3 then shows the Cumulative Distribution Function of DMM scores, together with a theoretical normal distribution (the solid lines). The theoretical normal distributions have the same mean and standard deviation as the scores in the data set. The dashed lines are the scores for all issues in the data set, while the dotted lines are the scores of issues that have a churn of at least 15.

Given the above results, the DMM enables ranking the maintainability of changes, despite our empirical data showing minor differences between open and closed source systems. Furthermore, we witness skewed distributions for some Code Properties. Our hypothesis is that some SIG-MM thresholds could be refined to suit the purpose of the DMM. For these two aforementioned reasons, additional research should provide empirical data to either confirm or refute our conjecture.

To the ends of detecting technical debt and helping developers identify its root-causes, the DMM yields result that can be leveraged in a straightforward manner. As the calculation example in Section V shows, the *DMM Score = 0.341* suggests that the overall change on average is poorly maintainable. Exploring the Delta Score results for the single properties is unequivocal as a developer can understand both *where* and *how much* her code is maintainable and address introduced technical debt. Taking as example Unit Size (*DS Unit Size = 0.09*) the change has low Score as only 9% of the Risk Profile Delta is low-risk. This enables developers to act towards improving the change maintainability, as extracting code in a new method and performing a call from the existing code will likely improve its maintainability. Smaller units are easier to understand, test, and maintain.

## VIII. DISCUSSION

Looking back at the assessment study, we are now in the position to discuss implications of DMM for further research and for applications in practice. We argue that our work progresses towards meeting two challenges in this context, *i.e.*, being able to *measure the maintainability delta of commits directly*, and *score and compare commits*.

### A. Practical implications

DMM is the first step towards an objective way to assess the maintainability of fine-grained code changes. Commits are the finest grain feasible in development practice today, creating many options for flexible aggregations matching practical use cases. For instance, commits can be aggregated into time periods (*e.g.*, into days, weeks, *etc.*), and developers can leverage tailored analysis to target technical debt at an early stage of development, rather than dealing with major refactoring resulting from accumulated bad code.

In the context of providing management insight into the maintainability impact of ongoing development, DMM provides a novel way to measure, score, and compare changes. Current system-level maintainability model struggle in assigning maintainability scores to fine-grained code changes, hindering root-cause analysis to the activities that caused maintainability risks that piles up as technical debt.

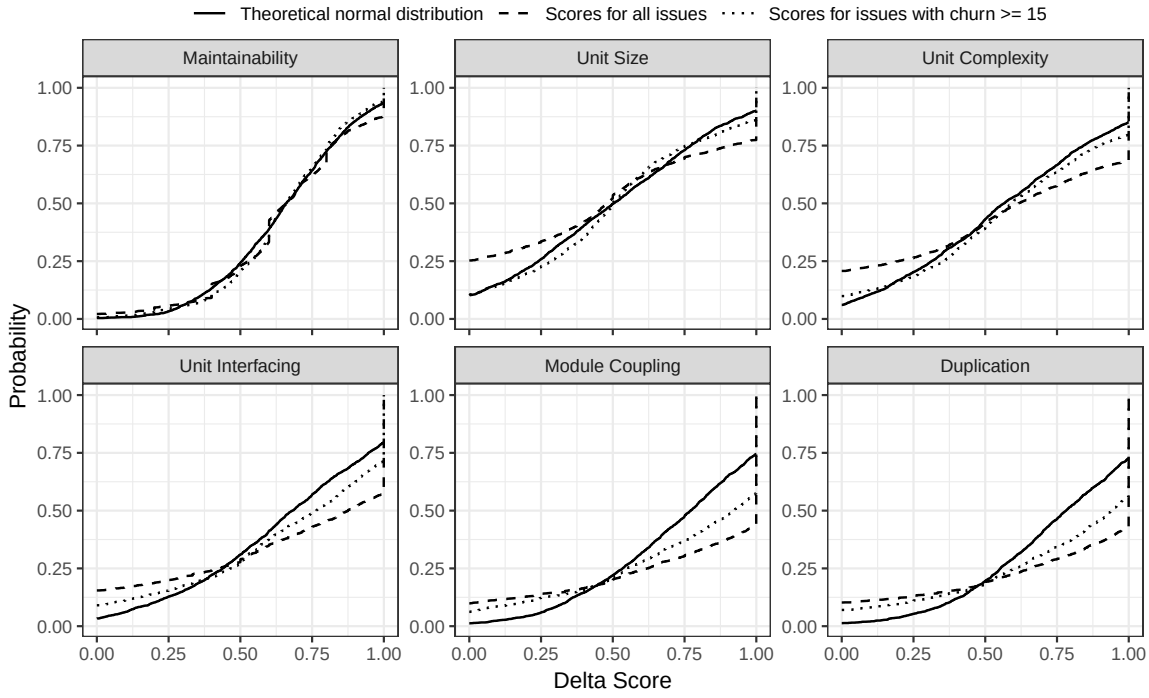



FIGURE 3. RQ2 – CUMULATIVE DISTRIBUTION FUNCTIONS FOR THE DELTA SCORE IN THE DMM.

DMM reuses concepts of the SIG-MM, therefore being largely technology independency as the SIG-MM and its underlying tooling currently support close to 200 programming languages. However, DMM still needs to deal with VCS's to analyze commits, for which we foresee new tools to be developed to feed the DMM with the required VCS change information. As this consists of mostly basic diff information, developing such tool support is not prohibitive.

### B. Research implications

The assessment study performed on DMM gives rise to follow-up projects aimed at improving DMM. Furthermore, we see various synergies with research in the direction of technical debt, and the socio-technical domain of providing feedback to software engineering teams.

From Figure 3, it is clear that not all code properties included in DMM produce the desired behavior of scores yet. In particular Unit Interfacing, Module Coupling, and Duplication, exhibit strong left skew in the scores. Many scores perfectly (*i.e.*, 1), indicating that the DMM is not yet sensitive or strict enough with those code properties. Further research will be needed to tune the thresholds at which code properties should consider committed code poorly maintainable. Currently, these thresholds are identical to those of the SIG-MM.

Another issue appeared in both the manual assessment and data analysis of code properties Module Coupling and Duplication: many commits contain hardly any duplicates or dependencies within the code they changed, leading to many perfect scores. In our assessment study the underlying coupling and duplication analyses were, for reasons of run time limitations, restricted to the files touched by a commit. Probably, more dependencies and duplicates would have been found in commits when analyzing entire code bases. This would imply the score distribution for these two code properties could, without the run time restriction, become more favorable.

The DMM's measurement of commits provides a fine-grained feedback method to developers to fix bad code creating technical debt. This opens up research directions asking for socio-technical methods into the effectiveness of fine-grained, and arguably, more timely, maintainability feedback to developers. Are developers more willing to act on commit-level feedback than on snapshot-based feedback? Does this lead in the long term to less technical debt? Does the direct link between the 'who, what, and when' that commits provide, combine well with maintainability feedback? Are there pitfalls to avoid in applying such metrics, *e.g.*, like "treating the metric" [35]?

## IX. CONCLUSION

The goal of this paper is to create a model that measures the maintainability of code changes and deal with fine-grained technical debt. To that end, we propose the *Delta Maintainability Model*, which measures the percentage of *low risk* lines of code in the change. The DMM bulids upon the concept of maintainability as proposed in the SIG Maintainability Model, focusing on metrics for Unit Size, Complexity, Interfacing, Duplication, and Coupling as underlying metrics.

We assess the extent to which the SIG-MM maintainability metrics remain meaningful for fine-grained measurements by manually analyzing 100 issues among four open source systems and contrasting the results with the Model Score. Our findings show that the DMM matches 67% of the manually assessed sample, and that most disagreements occur when the DMM is around 0.5. Our exploratory empirical analysis on

over 3000 issue fixing commits shows that DMM is suitable for ranking and comparing commits, with properties akin to that of a normal distribution.

As showed in Section V, the DMM can be used to measure the maintainability of a fine-grained software change such as an issue fix, or a pull-request. Furthermore, it provides actionable results for developers to address technical debt in the form of nonfunctional maintainability problems, with measurements that can be generalized across different software systems. Finally, the DMM Score can rank maintainability of changes allowing for direct comparison among different software changes.

Our initial analysis suggests that the DMM has strong possibilities to pinpoint technical debt code shortcomings by outlining the maintainability of individual change sets. In our future work, we will seek to refine the model and evaluate to what extent the *Delta* approach proposed can be generalized, for example to assess commits beyond bug fixes, taking teams, organizations, and packages into account. Furthermore, our intent is to put the model into practice, and report on our experiences.

**Acknowledgments.** This project received funding from European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 642954.

## References

[1] ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, Aug 2017.

[2] W. Cunningham. The wycash portfolio management system. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 29–30. ACM, 1992.

[3] Technical debt. https://martinfowler.com/bliki/TechnicalDebt.html.

[4] Technical debt quadrant. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html.

[5] N. A. Ernst, S. Bellomo, I. Ozkaya, R.L. Nord, and I. Gorton. Measure it? Manage it? Ignore it? Software Practitioners and Technical Debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 50–60. ACM, 2015.

[6] J.P. Miguel, D. Mauricio, and G. Rodríguez. A review of software quality models for the evaluation of software products. *arXiv preprint arXiv:1412.2977*, 2014.

[7] J.-L. Letouzey. The sqale method for evaluating technical debt. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 31–36. IEEE, 2012.

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

[9] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit. The Quamoco Product Quality Modelling and Assessment Approach. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1133–1142. IEEE Press, 2012.

[10] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[11] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works http://www.thoughtworks.com/ContinuousIntegration.pdf*, 122:14, 2006.

[12] G. Gousios, M. Pinzger, and A. van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[13] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. Di Penta, and A. Zaidman. Continuous delivery practices in a large financial organization. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 519–528. IEEE, 2016.

[14] T.J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, SE-2(4):308–320, Dec 1976.

[15] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Reliability and maintainability symposium, 2002. Proceedings. Annual*, pages 235–241. IEEE, 2002.

[16] P. Antonellis, D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis, C. Tjortjis, and N. Tsirakis. A data mining methodology for evaluating maintainability according to iso/iec-9126 software engineering–product quality standard. *Special Session on System Quality and Maintainability-SQM2007*, 2007.

[17] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, Jun 1994.

[18] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[19] T. L. Alves, J. P. Correia, and J. Visser. Benchmark-Based Aggregation of Metrics to Ratings. In *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pages 20–29, 2011.

[20] R. Baggen, J. P. Correia, K. Schill, and J. Visser. Standardized Code Quality Benchmarking for Improving Software Maintainability. *Software Quality Journal*, 20(2):287–307, June 2012.

[21] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. A probabilistic software quality model. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252. IEEE, Sept 2011.

[22] Sonarqube. https://www.sonarqube.org/.

[23] M. Conradt, L. Hinemann, B. Hummel, V. Bauer, and E. Juergens. A Framework for Incremental Quality Analysis of Large Software Systems. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 537–546. IEEE Computer Society, 2012.

[24] L. Heinemann, B. Hummel, and D. Steidl. Teamscale: Software quality control in real-time. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 592–595. ACM, 2014.

[25] Teamscale. https://www.cqse.eu/en/products/teamscale/overview/.

[26] E. Bouwers, A. van Deursen, and J. Visser. Evaluating Usefulness of Software Metrics: An Industrial Experience Report. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 921–930. IEEE Press, 2013.

[27] *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, March 2011. https://www.iso.org/standard/35733.html.

[28] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. In *ACM Sigplan Notices*, volume 41, pages 397–412. ACM, 2006.

[29] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

[30] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):2, 2008.

[31] M. di Biase, A. Rastogi, M. Bruntink, and A. van Deursen. The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes Technical Report. https://doi.org/10.5281/zenodo.2542535, 2019.

[32] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[33] J. Saldaña. *The coding manual for qualitative researchers*. Sage, 2015.

[34] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering*, 43(5):476–491, May 2017.

[35] E. Bouwers, J. Visser, and A. van Deursen. Getting what you measure. *Communications of the ACM*, 55(7):54–59, 2012.