2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)

Learning a Classifier for Prediction of Maintainability based on Static Analysis Tools

Markus Schnappinger
Technical University of Munich
Munich, Germany
schnappi@in.tum.de

Mohd Hafeez Osman
Technical University of Munich
Munich, Germany
osmanm@in.tum.de

Alexander Pretschner
Technical University of Munich
Munich, Germany
pretschn@in.tum.de

Arnaud Fietzke
itestra GmbH
Munich, Germany
fietzke@itestra.de

***Abstract*— Static Code Analysis Tools are a popular aid to monitor and control the quality of software systems. Still, these tools only provide a large number of measurements that have to be interpreted by the developers in order to obtain insights about the actual quality of the software. In cooperation with professional quality analysts, we manually inspected source code from three different projects and evaluated its maintainability. We then trained machine learning algorithms to predict the human maintainability evaluation of program classes based on code metrics. The code metrics include structural metrics such as nesting depth, cloning information and abstractions like the number of code smells. We evaluated this approach on a dataset of more than 115,000 Lines of Code. Our model is able to predict up to 81% of the threefold labels correctly and achieves a precision of 80%. Thus, we believe this is a promising contribution towards automated maintainability prediction. In addition, we analyzed the attributes in our created dataset and identified the features with the highest predictive power, i.e. code clones, method length, and the number of alerts raised by the tool Teamscale. This insight provides valuable help for users needing to prioritize tool measurements.**

***Index Terms*— Software Quality, Software Maintenance, Code Comprehension, Static Code Analysis, Maintenance Tools**

I. Introduction

Software vendors aim to develop software systems that fulfill all functional requirements, are economical to build in the first place while also being easy to maintain in the future. The largest share of the development costs is actually maintenance costs [1], [2]. Therefore, there exists a direct relation between the maintainability of a system and its economic profitability. Over time, maintenance costs increase as the code basis becomes larger and errors are propagated [3]. It is therefore critical for software vendors to establish continuous quality management to avoid cost explosions. Code reviews, for example, can help to evaluate and control the quality of source code [4]. Also, software health checks by external quality auditors are a tried and tested remedy [5], [6]. Though these manual inspection techniques are effective and well established, they are also expensive and time-consuming. Instead of continuously performing extensive and expensive reviews during development, many companies use static analysis tools to track the quality of their systems. These tools analyze source code and provide measurements about the program without actually executing it. Unfortunately, several studies confirm the high number of inadequate warnings emitted by these tools [7], [8]. Developers quickly feel overwhelmed by the large number of measurements provided by such tools. Thus, several approaches use polynomial functions to aggregate multiple measurements into one single numerical. One of the first approaches was the Maintainability Index introduced by Oman [9]. Several years later, Benestad et al. [10] point out the need to define a strategy for every metric-based assessment, consisting of well-defined selection-, combination-, aggregation- and interpretation techniques. Still, this strategy has to be defined manually, and especially the interpretation of the results requires enormous expertise. Instead of using a predefined polynomial with fixed weights, we try to capture that human intuition of professional quality experts with artificial intelligence. The company itestra brings 15 years of industrial experience to this joint research. In addition to software engineering projects, itestra also offers post-release software audits, sometimes referred to as software health checks [5].

Our research models the experience of professional quality analysts using machine learning. The goal is to establish an automated evaluation using metrics, that is based on expert judgment. This paper hence evaluates the following approach towards automated assessments: With the help of quality consultants working at itestra, we manually analyzed source code from three different projects, accounting for 115,373 lines of Java code. The experts labeled the corresponding classes with respect to their maintainability to create a labeled dataset. Then, we retrieved the output of three static analysis tools for these classes and attached the labels. Eventually, we trained supervised machine learning algorithms to assess the maintainability of source code based on static measurements. Analyzing the created dataset, we also identified the measurements with the highest predictive power. In our prediction experiments, we obtained promising results with respect to the achieved accuracy of 81% and F-Score of 80%. While we consider this a promising first step towards automated quality assessments, it is not yet sufficient for a stand-alone tool. However, our approach offers a valuable quick assessment for developers without access to professional, time-consuming software assessments. In addition, our feature evaluation showed that cloning information, Teamscale-Findings, and method length are the metrics with the highest correlation to maintainability as perceived by the experts.

978-1-7281-1519-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICPC.2019.00043

II. Related Work

Coleman et al. [11] investigated the relations between manual and metric-based automated assessments. Their study shows that the results of the automated maintainability analysis and the qualitative assessments performed by maintenance engineers strongly correlate. In their study, polynomial models were used to compare software systems. Benestad et al. [10] also predicted software maintainability based on metrics in their paper in 2006. Their approach mostly considered measurements describing the relation of one class to other classes of the system, e.g. coupling. While we use machine learning to capture the experience of professionals, they discuss different selection-, aggregation-, combination- and interpretation methods to deduce the maintainability. In [8], Yüksel and Sözer applied machine learning techniques to classify alerts emitted by static analysis tools. In contrast to our work, these alerts are bug-related and do not consider possible maintainability issues. Koc et al. [7] follow a similar approach and focus on bug alerts, too. In a first step, they isolate the code that was highlighted by the analysis tool. Applying machine learning to these code snippets, they either confirm or refute the finding. Hegedűs et al. [12] proposed an approach to build a method-level maintainability prediction model based on human evaluation. Three surveys were conducted resulting in three datasets of source code maintainability. They found that none of the datasets was suitable to build a reliable regression model.

Since labeling source code is challenging and time-consuming, Kumar et al. referred to the number of changed lines per file to quantify maintainability [13], while Hegedűs et al. conducted a survey to collect maintainability labels [12]. In contrast to these works, our study refers to maintainability as perceived by professional quality consultants. In summary, we capture the maintainability perception of quality experts using metrics. In contrast to Oman [9] and Coleman [11], we do not aggregate the metrics using polynomial functions with fixed parameters, but apply machine learning algorithms to learn the expert evaluation. Opposed to work from Hegedűs et al., this research focuses on class-level maintainability.

III. Experiment Setup

In contrast to other studies, this paper does not measure task-completion-time to refer to comprehensibility [14] or the number of revisions to refer to maintainability [13], [15]. Instead, we work together with professionals from industry and their experience-based definition of maintainability. For this purpose, we define maintainability as the *ease of change*, leading to two sub-characteristics:

1) As a developer, can I understand what the code does and identify where certain aspects are implemented?
2) As a developer, do I have to worry about hidden dependencies of the code I am currently modifying?

While the first aspect addressed the need to comprehend the source code, the second one focuses on where else the developer has to apply changes. For example, duplications of the code snippet have to be found and modified as well.

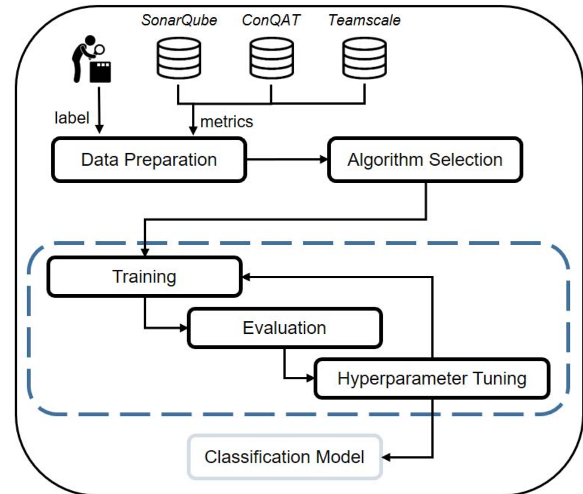

Fig. 1. Overall Approach

Provided the expert judgment, this research answers the following questions:

- Is it possible to predict a human intuition of the maintainability of source code based on tool measurements?
- Are there relations between metrics and expert judgment, and which metrics have the highest predictive power?

A. Overall Approach

Figure 1 depicts the overall framework of our approach. In the data preparation phase, we extracted metrics from the code sample using static analysis tools, performed data cleaning and combined the metrics and the label. Next, we train and validate the models. We selected a diverse set of 21 algorithms representative for different approaches. For each classifier, the iteration *train* $\rightarrow$ *evaluate* $\rightarrow$ *parameter tuning* continued until all possible parameter combinations were evaluated. At this stage, we also analysed the predictive power of the features.

B. Study Objects

To evaluate the approach, a dataset of source code and its evaluation has to be created. We took our sample from three software systems written in Java. The chosen sample includes 115,373 Lines of Code (LoC), distributed over 345 classes. To ensure a high diversity among the study objects, we chose one small project with approx. 45k LoC, one medium-sized system with around 380k LoC and one big project with more than 3M LoC. The age of the systems lies between 4 and 19 years. The projects cover in-house, off-shore, and open-source development. Two of the systems are industrial projects located in the insurance domain. The third system is the software testing framework JUnit 4 (Version 4.11). Table I provides an overview of the systems.

C. Static Analysis Tools for Data Collection

Static code analysis tools analyze source code without actually executing it. Their measurements serve as input for

TABLE I
ANALYZED SOURCE CODE

	System A	JUnit 4 (4.11)	System C
Domain	*Insurance*	*Software Dev.*	*Insurance*
Purpose	*Offer Management System*	*Testing Framework*	*Damage Evaluation System*
First Release	*2000*	*2014*	*2014*
Development	*Outsourced*	*Open-source*	*In-house*
Size	*3.1M LOC*	*44.6k LOC*	*380k LOC*
Chosen Sample	*65k LOC* *160 Classes*	*10k LOC* *75 Classes*	*41k LOC* *110 Classes*

our experiments. We targeted to use both commercial and free-to-use tools and to integrate both basic measurements as well as complex metrics. Among the various available tools we chose the following three:

- *ConQAT:* The tailorable, open-source framework integrates clone detection and structural assessments [16].
- *Teamscale:* This commercial tool evaluates both structural properties and code style to identify code anomalies. These anomalies are called findings and are automatically categorized according to their severity [17].
- *SonarQube:* The open-source tool offers tailorable quality gates. It also provides aggregated measurements like code smells and potential vulnerabilities or bugs [18].

Examples of the extracted attributes are the following:

- *Size*: Lines of Code, Source Lines of Code, Method Lines of Code, Number of Conditions
- *Structural*: Max. Method Length, Avg. Method Length, Max. Block Depth, Loop Length, Max. Loop Depth
- *Cloning*: Clone Coverage, Clone Units
- *Complex Measurements:* Cognitive Complexity, Code Smells, Teamscale-Findings (i.e. the number of quality violations identified by Teamscale)

D. Labeling

In order to learn from our code base, the source code is analyzed and labeled by experts. On one hand, it is impossible to evaluate source code without context. On the other hand, we had to draw a line what to take into account and what to omit from the analysis. Hence, we chose a class-level granularity. The possible classification is threefold: *A, B*, and *C*.

- **Label A** indicates the absence of indicators for maintainability problems with respect to the ease of change.
- **Label B** covers classes with some room for improvement.
- **Label C** is assigned to code that is clearly hard to maintain and requires high effort to be changed.

Our experiment aims to capture the experience of professional experts. Therefore, it is imperative to label the data according to that expertise. Although this limits the size of the dataset, we still managed to label 345 classes, representing more than 115k Lines of Java Code that had to be inspected and evaluated. In this context, it is not advisable to automatically label large datasets with, for example, a rule-based script. The machine learning algorithm would not capture the expert opinion, but would simply reverse engineer the rules used for

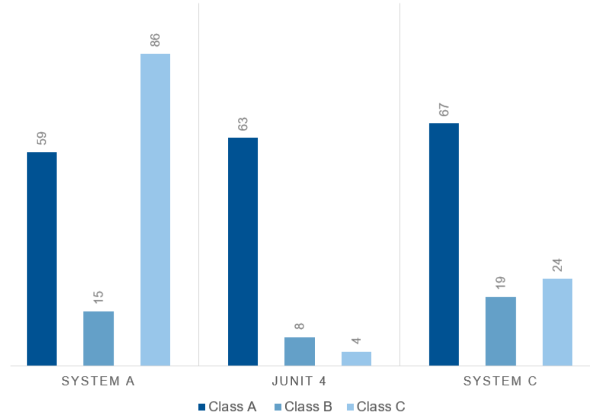

Fig. 2. Label distribution in each system

the automated labeling. During the joint assessment of the study objects, both the quality consultants and the researchers evaluated the source code. The judgment of the researchers was then discussed in joint validation sessions, ensuring the provided label matched the opinion of the experts. The labeling procedure resulted in 182 instances out of 345 (52.75%) being assigned label A, 51 instances (14.78%) assigned label B, and 112 instances (32.46%) are categorized as C. The distribution of the labels among the single projects is shown in Figure 2.

IV. EXPERIMENT

Though our dataset covers more than 115k Lines of Code, it accounts for just 345 instances. To avoid bias introduced by splitting the 345 data points in fixed training, validation, and test sets, we used 10-fold stratified cross-validation. Since we are using a threefold label and thus face a multiclassification problem, we use accuracy, precision, recall, and F-Score to evaluate the performance of the algorithms as suggested by Sokolova [19]. In addition, we analyzed differences in the performance between the classes A, B, and C.

A. Prediction Results

Our experiments are implemented using the Waikato Environment for Knowledge Analysis (Weka) [20] Version 3.9.3. Every algorithm was run once in its default configuration before hyperparameter optimization was applied. The results discussed in the remainder of this subsection correspond to the best observed performance of each classifier.

The algorithm with the best results was J48, a decision tree based algorithm. It was able to classify 279 instances (81%) correctly. It achieved a precision of 79.7% with a recall of 80.9%, combining for an F-Score of 80.1%. The performance of the best classifiers and baseline comparisons are denoted in Table II. The table also shows that J48 outperforms the other classifiers in all four performance measures.

In addition to the performance over the whole dataset, we also investigated differences between the categories. Table III denotes the F-Score per class for the three best-performing classifiers. Indeed, a significant drop for files from category B

TABLE II
EXPERIMENT RESULTS

Classifier	Accuracy	Precision	Recall	F-Score
J48	0.8087	0.7967	0.8087	0.8009
LMT	0.7971	0.7693	0.7971	0.7757
SimpleLogistic	0.7971	0.7577	0.7971	0.7566
...	...	...	...	...
OneR	0.7102	0.6430	0.7101	0.6540
...	...	...	...	...
Multilayer Perceptron	0.6667	0.6667	0.6667	0.6667
ZeroR	0.5275	n/a	0.5275	n/a

TABLE III
RESULT DIFFERENCES

Classifier	F-Score		
	Category A	Category B	Category C
J48	0.874	0.449	0.842
LMT	0.859	0.290	0.862
SimpleLogistic	0.860	0.161	0.860

can be observed. J48 only achieves an F-Score of 44.9%, while the performance is even worse for LMT and SimpleLogistic with 29.0% and 16.1%, respectively.

B. Attribute Evaluation

Given the combination of the tool output and the categories assigned by manual inspection, we analyzed the resulting matrix for the most influential features. We applied six different feature selection algorithms on our data. Table IV lists the results of the algorithms *InfoGain* and *OneR Attribute Evaluation*. Due to the space limitations of this paper, we list the results of neither all algorithms nor all 67 attributes. For presentation reasons, we count the number of times a feature was part of the 10 highest-ranked attributes. This number is provided in the right-most column of Table IV. Features with less than four votes are omitted from the table. The attribute evaluation identified clone coverage as one of the most predictive features. Also, clone units were selected by five out of six techniques, whereas Teamscale-Findings and the maximum size of a method are selected in four of the six cases. Hence, these characteristics are considered the most influential features.

TABLE IV
MOST INFUENTIAL FEATURES

Attribute	InfoGain Score	OneR AttrEval	Top10 Appearances
Clone Coverage 50NN[(i)]	*0.3070*	*69.86*	*6*
Clone Units 50NN[(i)]	*0.2633*	*71.01*	*5*
Teamscale-Findings	*0.2777*	*66.38*	*4*
Max. SLOC per Method	*0.2415*	*64.03*	*4*
Max. LOC per Procedure	*0.2226*	*65.80*	*4*
Max. LOC per Method	*0.2164*	*65.51*	*4*

[(i)] non-normalized, minimum length of 50 units

V. DISCUSSION

This experiment uses a threefold label as we think a threefold classification captures the expert understanding of maintainability best. We did not compare the results with other labels such as a twofold label. Binary labels do not reflect the real world, and, even more importantly, do not reflect the way experts perceive quality. For the very same reason, we decided to use a classification model instead of regression models as implemented in [12]. From our experience, a numerical value does not correspond to the way experts perceive quality. Quality analysts do not target to retrieve a numerical value but aim to develop a general understanding of existing problems.

A. Interpretation of the Prediction Results

The results presented in Section IV show that the assigned label corresponds to the experts' categorization in up to 80.87% of the time. The classifiers J48, LMT, and SimpleLogistic delivered the best results in our experiment. They clearly outperform baseline-classifiers such as ZeroR by a large margin. The best-performing algorithm, J48, is based on C4.5, a decision tree algorithm described in detail in [21]. It achieved an accuracy of more than 80% and an F-Score greater than 80% as well. LMT, the second-best performing algorithm, also implements a decision tree. In contrast to J48, LMT uses logistic functions at the leaves [22].

Analyzing the performance of these three algorithms, we observed significant differences between the three categories A, B, and C. As illustrated in Table III, the F-Score for category B just ranged from 16% to 45% while being above 84% for all other classes. We interpret this finding as follows. Our prediction approach is able to identify classes with good quality and classes with bad quality. It performs poor for mediocre labels. To solidify this interpretation, we analyzed the false positives. Using J48, 7 instances of category A were erroneously classified as C (4%), 12 instances were mistaken for B (7%), and 163 instances (89%) were labeled correctly. In contrast, 10 instances of category C were misclassified as A (9%), 6 instances (5%) were classified as B and 96 instances (86%) were labeled correctly.

Given these observations, the classification results can be interpreted both optimistically and pessimistically. The goal of industrial software quality assessments is to identify which parts of the system suffer from bad quality. Based on the identified issues, measures are taken whether to rebuild the system, renovate certain components or restructure the development team [6]. The analysis of the false positives shows that the automated approach is not yet suitable to replace the human expert in finding these trouble spots. Not only does it assign wrong labels in 19% of the time, but the produced false positives are actually severe. Hard to maintain code was misclassified as easy to maintain in 9% of the times. Missing that number of potential trouble spots prohibits to rely on the classification in critical software assessments. System owners should not derive far-reaching actions based on a classification with just 81% accuracy.

However, we still consider the achieved results the first step towards automated quality analysis. Static analysis tools are not only used for external quality assessments, but also for continuous quality control. Using the tools SonarQube, Teamscale and ConQAT, one obtains 67 different measurements, making it hard to reason about maintainability at a glance. This work presents a method to aggregate different metrics in a way that is learned from experienced experts. Though it is not comparable with human experts, the automated classification helps developers to identify a great share of the code with maintainability issues.

B. Interpretation of Attribute Evaluation

As mentioned earlier, static code analysis tools analyze source code and report the measured characteristics. The user must draw conclusions and interpret the metrics based on his experience and expertise. In this research, we created a dataset of source code, the static tool output for this code and its expert evaluation. The most influential metrics presented in Table IV now allow developers insights into the expert evaluation. Hence, we believe our feature analysis provides valuable guidance for developers which metrics to focus on to predict the expert opinion. The first two metrics to be taken into account are cloning coverage and clone units since they have the highest correlation with the expert judgment. Then Teamscale-Findings should be respected, as well as the maximum method length. This does not mean that all other metrics should be ignored, but this set already offers a good indication of code maintainability.

In the context of this research, maintainability was defined as the ease of change, i.e. a combination of comprehensibility of the code itself, and understandability which dependencies have to be updated as well. Clone coverage and clone units refer to code duplications. Modification of a code snippet with a duplicate in another place forces the developer to search for the clone and apply the change here as well [23]. With code duplications hence leading to decreased maintainability, it is not surprising that cloning measurements show high predictive power. Interestingly, as opposed to the size of a method, the size of a class is not amongst the most influential features. Teamscale and other static analysis tools automatically rate large classes with more than 750 Source Lines of Code as hard to maintain [24]. We did not apply such fixed thresholds and actually rated every program class manually. Hence, we consider the results of the feature analysis a valuable contribution to research, as it reverse engineers the intuition of the human experts.

C. Threats to Validity and Future Work

In this study, the maintainability of classes was evaluated manually. To mitigate the threat to internal validity, validation sessions were performed to discuss the evaluation. Still, the assessment was performed by quality consultants from just one company. We notice that our dataset consists of only three systems, covers just two domains and only includes Java code. Also, the used dataset is imbalanced with class A and class C dominating the data distribution.

This work presents initial findings and promising results on using static analysis metrics to classify the maintainability of source code. For future work, we plan to explore the possibility of using metrics derived from mining identifiers, method names, and comments as well. Also, investigating the influence of class network metrics on maintainability is part of our future plan. In the meantime, reducing the number of features and increasing the size of the dataset is our priority in order to reduce the risk of overfitting and increase the reliability of the classification model. Finally, there is one major limitation to the chosen approach. While inspecting and evaluating the source code, we observed that several negative findings are of semantic nature. While static code analysis tools have their strengths in assessing structural characteristics, they cannot detect semantic flaws. For example, discrepancies between implemented behavior and documentation lead to lower perceived comprehensibility but are not reflected by structural metrics.

VI. CONCLUSION

The goal of this study is to model the experience of professional quality analysts using machine learning. Therefore, a sample of 115,373 Lines of Code was selected from three study objects, including two industrial systems. In joint work with professional quality analysts, the source code was inspected and evaluated on class-level. The evaluation is based on the ease of change, i.e. the comprehensibility of the source code, and the comprehensibility which external dependencies have to be updated after a change. The experts assigned the labels *A*, *B*, and *C* to each file. Label C indicates the code is hard to maintain, while A corresponds to the absence of indications for low maintainability. While manual assessments are a well-established method to evaluate the quality of software, many developers use static analysis tools to monitor quality. In this study, we used metrics emitted by such tools to learn and predict expert judgement. The algorithm J48 achieved an accuracy of 81% and a precision of 80%. We consider this approach to be a promising first step toward automated software evaluation. While the performance is sufficient for quick assessments, it is not yet suitable to replace an expert review. In addition, we analyzed the used features and investigated their predictive power. We found that clone coverage and clone units are the most influential features. Teamscale-Findings, i.e. the number of identified quality violations as computed by Teamscale, and the maximum method length also have high predictive power. This result provides guidance which metrics to prioritize for maintainability evaluations, based on the correlation with the expert judgment.

ACKNOWLEDGMENT

The authors would like to thank itestra for the participation and their commitment to publish this joint work. In particular, we appreciate the valuable and constructive knowledge transfer and the opportunity to learn from industrial experience.

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[2] B. Boehm, J. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 592–605.

[3] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[4] S. McIntosh, Y. Kamei, B. Adams, and A. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[5] M. Pizka and T. Panas, "Establishing economic effectiveness through software health-management," in *1st International Workshop on Software Health Management, Pasadena*, 2009.

[6] M. Schnappinger, M. H. Osman, A. Pretschner, M. Pizka, and A. Fietzke, "Software quality assessment in practice: a hypothesis-driven framework," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2018, p. 40.

[7] U. Koc, P. Saadatpanah, J. Foster, and A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 2017, pp. 35–42.

[8] U. Yüksel and H. Sözer, "Automated classification of static code analysis alerts: a case study," in *29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 532–535.

[9] P. Oman and J. Hagemeister, "Construction and testing of polynomials predicting software maintainability," *Journal of Systems and Software*, vol. 24, no. 3, pp. 251–266, 1994.

[10] H. C. Benestad, B. Anda, and E. Arisholm, "Assessing software product maintainability based on class-level structural measures," in *International Conference on Product Focused Software Process Improvement*. Springer, 2006, pp. 94–111.

[11] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *IEEE Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[12] P. Hegedűs, G. Ladányi, I. Siket, and R. Ferenc, "Towards building method level maintainability models based on expert evaluations," in *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*. Springer, 2012, pp. 146–154.

[13] L. Kumar, A. Krishna, and S. K. Rath, "The impact of feature selection on maintainability prediction of service-oriented applications," *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 137–161, 2017.

[14] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 31–40.

[15] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings of Eighth IEEE Symposium on Software Metrics*. IEEE, 2002, pp. 87–94.

[16] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka, "Tool support for continuous quality control," *IEEE software*, vol. 25, no. 5, pp. 60–67, 2008.

[17] 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*. ACM, 2014, pp. 592–595.

[18] G. Campbell and P. P. Papapetrou, *SonarQube in action*. Manning Publications Co., 2013.

[19] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[21] R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[22] N. Landwehr, M. Hall, and E. Frank, "Logistic model trees," *Machine Learning*, vol. 95, no. 1-2, pp. 161–205, 2005.

[23] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE 2009. IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 485–495.

[24] CQSE GmbH, *Teamscale, the Official User Reference, Version 4.8*, Munich, Germany, 2019.