Revisiting Inter-Class Maintainability Indicators

1st Lena Gregor
Software and Systems Engineering
Technical University of Munich
Munich, Germany
lena.gregor@tum.de

2nd Markus Schnappinger
Software and Systems Engineering
Technical University of Munich
Munich, Germany
markus.schnappinger@tum.de

3rd Alexander Pretschner
Software and Systems Engineering
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

***Abstract*—Over the last few decades, a variety of static code metrics have been published and promoted to measure the maintainability of software systems.**

This study evaluates 12 common static code metrics for their correlation with observed maintenance efforts. Leveraging modern repository mining techniques, we examine the historical data of three large open-source software systems with a combined size of over 1M LOC and over 10k classes. We automatically identify maintenance activities and measure the effort needed to perform them through revised lines of code. Then, we investigate if the state of the system as captured by these metrics is an indicator for the required maintenance effort.

In contrast to earlier research, our results could not validate a general correlation between any of the examined metrics and maintainability. Instead, all evaluated metrics showed positive *and* negative correlations with maintenance effort depending on the considered time interval. Strong correlations only hold for specific projects, and within these projects, only for limited time spans. Across the project history, however, all metrics showed moderate correlations at most.

As no metric was found to be a good indicator for high maintenance efforts in all contexts, we advocate against using any of the evaluated metrics without project-specific validation. If metrics are to be used to monitor the maintainability of a system, either directly or through models based on these metrics, engineers have to validate their applicability not just for the project at hand, but also for the current time span.

***Index Terms*—software maintainability, maintainability prediction, static code metrics, repository mining**

I. Introduction

Several static code metrics have been proposed and used in the last decades to assess the maintainability of software systems. Prominent examples are Halstead's metrics [1] and the Chidamber-Kemerer metric suite [2]. However, only a few of these metrics have been sufficiently validated [3], [4]. As Sharma and Spinellis observe [5], several code metrics fail to capture the quality attribute they aim to describe [5]. Furthermore, a study in the context of defect prediction has shown that single static code metrics predict quality attributes differently well for different systems [6].

One promising approach to finding a universal predictor for maintainability may be through aggregating code metrics based on machine learning as in [7]. However, their models predict the maintainability judgment of analysts that focuses on the quality of individual code files. Thus, only metrics quantifying properties within program classes (intra-class metrics) showed a correlation with their maintainability label.

There are indicators in the literature that metrics about the relationships between program classes (inter-class metrics) can also be good predictors [8]–[11] for maintainability. For single inter-class metrics to be good predictors, they need to highly correlate with maintainability. When multiple metrics are aggregated e.g. through machine learning approaches, they must at least offer some correlation with maintainability. That influence does not necessarily have to be large, but should at least be reasonably constant within a project. Otherwise, the applicability of the metrics is limited.

However, the results reported in related work are equivocal. For instance, Dagpinar and Jahnke [8] found a high positive correlation between efferent coupling metrics and maintainability measures. Whereas Sjøberg et al. [9] found a high negative correlation between coupling and maintainability. Considering the influence of inheritance on maintainability, Daly et al. [10] found maintenance tasks were performed faster in a system with an inheritance depth of three compared to a system with a depth of zero. Contrarily, Cartwright and Shepperd [11] found that changes in a system with an inheritance depth of zero were performed faster than in a system with a depth of three. The results of Prechelt et al. [12] and Dagpinar and Jahnke [8], however, suggest that inheritance depth has no significant influence on the maintenance effort.

Additionally, the studies have the following limitations: First, most of them were conducted on relatively small systems, the largest consisting of 430 program classes, and often referred to maintenance activities specifically designed for the study, which could have influenced their results. Second, software systems have evolved significantly over the past decades. Therefore, it is unclear whether findings from the early stages of object-oriented programming still apply to contemporary systems. Hence, as of now, it remains speculative if inter-class metrics are suitable predictors for the software maintainability of contemporary software systems.

Solution: We investigate the relationship between maintenance effort and various single inter-class metrics in depth. We analyze three large open-source software systems which total over 1M LOC. Leveraging modern repository mining techniques, we automatically identify and classify past maintenance activities and measure the associated effort via the Revised Lines of Code (RLOC). In particular, we want to answer the following question: Do inter-class metrics reliably correlate with maintainability? A metric that reliably correlates with maintainability ideally shows correlations …

1) consistently across different projects for the same maintenance activity
2) consistently within one project for different maintenance activities
3) consistently over time within a project

Hence, our analysis covers several dimensions: We analyze the metrics on multiple projects. Per project, we analyze the relationship across the whole development history and investigate if relationships manifest only during specific time intervals. Additionally, we collect data on two granularity levels: commit-level and task-level.

Contribution: Our results indicate there is no general correlation between any of the examined code metrics and the observed maintainability effort. None of the metrics showed consistent results for each of the software projects. Therefore, a metric that is a good measure of maintainability in one project is not necessarily a good measure in another project.

However, the strength and direction of the correlation not only varied between different projects but also between different time intervals and maintenance activities within the projects. Our results show that even if a metric strongly correlates with maintenance effort within one project at a specific point in time, the correlation can change drastically over time. Additionally, the type of maintenance activity also influences the correlation. This means that a metric that reliably predicts the effort for one type of maintenance activity does not necessarily predict the effort for another type of maintenance. We, therefore, conclude that single inter-class metrics should not be used as surrogates for maintainability without frequent context-specific validation.

II. Extracting Maintenance Effort

The goal of this study is to investigate the relationship between internal software attributes and the maintainability of a system. Thus, a qualitative or quantitative maintainability assessment is necessary. Maintainability describes the expected or observed effort needed to perform maintenance activities. In this study, we refer to historical data available in the repositories of the analyzed systems. First, we identify maintenance activities that have been performed on the system. Second, we classify these maintenance tasks according to Swanson's dimensions of maintenance, i.e. adaptive, corrective, and perfective maintenance [13]. Third, we associate the tasks with the effort required to perform them, which we extract from the repositories as well.

A. Identifying Maintenance Activities

For this study, it is essential to distinguish between different development activities. In all analyzed projects, issue trackers are used to plan and coordinate all implementation tasks. This allows us to leverage project-specific tags to identify and categorize maintenance activities. For each of the chosen projects, we manually evaluated each of the provided tags and their descriptions and mapped them to the maintenance types by Swanson [13]. Issues with tags such as 'bug' or 'bugfix' are considered corrective maintenance tasks, while tags including 'performance', 'cleanup', and 'refactoring' hint towards perfective maintenance. Issues tagged with 'enhancement' either match adaptive or perfective maintenance work. As we cannot discriminate the two types based on this tag, we combine them into one category. Thus, our analysis distinguishes only two maintenance types: *adaptive/perfective* and *corrective* maintenance. Table I summarizes which tags we used for each project to identify the maintenance types as well as the number of issues with and without tags.

TABLE I
Tags indicating maintenance work for each of the projects.

Project	**Adaptive/ Perfective**	**Corrective**	**#Issues with Tag**	**#Issues without Tag**
Artemis[a]	enhancement, code quality, performance, refactoring	bug, bugfix	2,692	397
RxJava[a]	Enhancement, Cleanup, Performance, Performance allocation	Bug	2,220	1,436
Spring-Framework[b]	type: enhancement	type:bug	1,444	1,984

[a]Values retrieved on 25.11.2021.
[b]Values retrieved on 06.01.2022.

B. Quantifying Maintenance Effort

The nature of this study requires an objective way to capture maintenance effort, which also allows for collecting a large number of data points automatically. Expert assessments and manually-labeled maintainability datasets as proposed in [14] and [15] are inapplicable in this study, as the manual labeling process does not scale to our use case. Some studies measure the time needed to perform a given task on the software system [9], [10]. This metric is objective and can be measured with little overhead, but is usually not publicly available. In [16], the maintenance effort is measured by the number of revisions a class went through in a specific period of time, which can be retrieved through public commit histories. However, we follow the approach of measuring the number of RLOC [9], [11], [16] as this is more fine granular. One limitation of this measure is the underlying assumption of approximately equal implementation effort for each line of code. However, RLOC is found to be correlated to the time needed to perform maintenance work [17]. Furthermore, it can be retrieved easily and fully automated and is widely accepted in this area of research.

III. Analyzed Inter-Class Attributes

A. Inheritance

Inheritance is a concept in object-oriented programming where classes can have superclasses from which they inherit certain characteristics. The results in [9]–[11], [15] indicate there is an influence of inheritance depth on the maintainability of software systems. However, it is unclear what this influence looks like, e.g., Daly et al. [10] found that maintenance tasks were performed faster in a system with an inheritance depth of three versus a system with a depth of zero, while Cartwright and Shepperd [11] found that changes in a system with inheritance depth of zero were performed faster than in a system with a depth of three.

The most commonly applied metrics to measure the inheritance structure of classes in a software system are Depth of Inheritance Tree (DIT) and Number of Children (NOC) proposed by Chidamber and Kemerer [2]. However, the DIT metric is criticized for holding ambiguities when multiple inheritances and multiple root classes are combined [18]. Furthermore, the NOC metric is criticized for not fully fulfilling its theoretical basis [18]. Thus, Li [18] proposed improved versions of these measurements: The Number of Ancestor Classes (NAC), which is the total number of classes from which a class inherits, and the Number of Descendent Classes (NDC), which describes the total number of sub-classes of a class. We use these two metrics to capture the inheritance relationships of a program class in this study.

B. Coupling

Coupling is the manner and degree of dependencies between software classes [19]. There are many different variations of metrics that consider various forms of coupling between classes in the literature. Briand et al. [20], identified three different types of coupling: Class-Attribute, Class-Method, and Method-Method. The Coupling between Object Classes (CBO) metric proposed by Chidamber and Kemerer [2] only considers coupling through the use of a method or instance variable from another class. Li [18] distinguishes between Coupling Through Abstract Data Type (CTA) and Coupling Through Message Passing (CTM). Martin [21] furthermore considers the direction of a coupling with afferent and efferent coupling. For *afferent* coupling, all classes that use and depend on the class are considered, whereas for *efferent* coupling, all classes that the class uses and, therefore, depends on are considered. In 2013, Yamashita and Moonen [22] found an association between afferent and efferent coupling and difficulties when performing maintenance tasks. A study by Dagpinar and Jahnke [8] in 2003 found a strong correlation between efferent coupling and maintenance effort. However, no significant correlation was found for afferent coupling [8]. In this study, we consider the following types of dependencies:

1) Coupling through Variable: If class A has an attribute or a variable of type B, then classes A and B are coupled through this variable.
2) Coupling through Parameter: If a method in class A has a parameter of type B or returns a value of type B, classes A and B are coupled through this parameter.
3) Coupling through Method: If class A uses a method of class B, classes A and B are coupled through this method.

Each of these three relations is once considered from the efferent and afferent perspective. This leaves us with six coupling metrics in total: Afferent Coupling through Variable (ACV), Afferent Coupling through Parameter (ACP), Afferent Coupling through Method (ACM), as well as Efferent Coupling through Variable (ECV), Efferent Coupling through Parameter (ECP), and Efferent Coupling through Method (ECM).

C. Code Duplication

Code duplicates, often called code clones, are pieces of code that are similar with respect to a definition of similarity [23]: *"[...] [Similarity] can be based on text, lexical or syntactic structure, or semantics"* [23, p. 2]. Intuitively, code duplication increases the chances that, if one instance of the clone is changed, developers need to also apply the change in all of its siblings. This increases the maintenance effort for program classes containing clones. Every code snippet with a duplicate at a different location is considered a code clone. The ratio of duplicated code in a software system or class is commonly referred to as its clone coverage. All instances of code that are duplicates of each other form a clone class.

Juergens et al. [24] showed that inconsistent changes to clones are very frequent and that a significant number of faults are induced by these changes, which impedes bug fixing and, therefore, increases the maintenance effort even more. Monden et al. [25] found that code with code clones is subject to more changes than files without clones. This indicates lower maintainability for files with code clones compared to files without code clones. More instances of code clones could further increase the need for more revisions compared to fewer code clone instances. Kapser and Godfrey [26], however, analyzed two systems and found that up to 71% of code clones are considered to have a benign impact on the maintainability of the systems. In their study, clones were deliberately used to flatten the inheritance hierarchy and facilitate testing activities.

There are multiple different definitions for code clones and approaches for clone detection. We use Teamscale[1], which implements statement-based clone detection [27]. Thus, comments, white spaces, and variable names are allowed to differ between duplicates. We used the default option that regards clones with at least 10 common consecutive statements.

In total, we consider four cloning-related code metrics in our analysis: (1) Number of Clones (NC), the total number of clone instances in a program class. (2) Number of Clone Classes (NCC), the number of different clone classes in a program class. (3) Number of Cloned Lines (NCL), the total number of lines in a program class that are cloned code. (4) Clone coverage (CC), the ratio of lines of code in a program class, which are part of a clone.

[1]Teamscale is a commercial static analysis tool provided by CQSE GmbH. More information can be found on the company's website: https://www.cqse.eu/en/teamscale/overview/

TABLE II
STUDY OBJECTS DESCRIPTION (JAVA FILES ONLY)

Project	Created on	# Developers	# Files	# LOC
Artemis	29.09.2016	122[a]	1,095[a]	177,184[a]
RxJava	08.01.2013	283[a]	1,870[a]	468,957[a]
Spring-Framework	08.12.2010	578[b]	7,217[b]	420,156[b]

[a]Value retrieved on 25.11.2021.
[b]Value retrieved on 06.01.2022.

IV. Study Design

A. Study Objects

Our study objects comprise three large open-source projects: *Artemis*[2], *RxJava*[3], and *Spring-Framework*[4]. Our selection criteria respect several aspects: (1) We only include projects mainly written in Java. As different programming languages offer different syntactical features, an analysis of the structural metrics across languages might distort the results. (2) Our study requires access to the commit histories and the issue trackers of the study projects. To foster the reproducibility of our study, we considered only open-source projects with publicly available issues and development history. (3) All projects are hosted on *GitHub*. This allows automated querying and extracting of data via an advanced API. (4) The projects need to allow the automatic identification of maintenance tasks. The chosen projects use tags to differentiate various maintenance and development activities. (5) To provide a large data corpus, the systems should be large in size and actively maintained over several years. The study objects contain between 177k and 469k LOC in 1.1k to 7.2k Java classes. In contrast, the largest system used by related work contains only 430 Java classes [28]. A summary of the projects and their metadata is provided in Table II. To avoid data from large projects overshadowing data from smaller projects [29], we analyze each of the systems individually.

B. Study Setup

With this study, we investigate the relationship of inheritance, coupling, and cloning metrics with the maintainability of three large software systems. For all systems, we extract issues from their issue trackers that are associated with maintenance work. A summary of the tags we used to identify and categorize maintenance tasks for the different projects of our study is provided in Table I in Section II-A.

For each issue, we analyzed all commits associated with the issue and computed the maintenance effort, i.e. the RLOC of each commit. Then, we calculated the code metrics based on the *parent* of this commit. This captures the state of the system at the time the maintenance activities started.

Unfortunately, there is no static code analyzer offering implementations for all considered metrics. Therefore, we utilize different tools for different measurements. For calculating the cloning metrics, we used the clone detection of Teamscale[5], which provides a history of all added and removed clones for the full lifecycle of a project. Based on that we determined the clones present at the respective commit and calculated the metrics with our own implementation. Although Teamscale is a commercial tool, free licenses are available for research purposes, thus ensuring the reproducibility of our results. All other considered metrics were implemented based on the *CK* project[6], an open-source tool with implementations for many software attribute metrics, publicly available on GitHub. We followed the descriptions of the original authors [18], [20], [21].

To measure the correlation between the code metrics and the maintenance effort, we refer to Spearman's rank correlation coefficient [30]. It is a non-parametric correlation measurement used to describe the monotonic dependency between two random variables. In contrast to Pearson's correlation coefficient, it does not assume a normal distribution of the analyzed data. The Spearman coefficient is defined within *[-1; 1]*, with larger values indicating stronger correlations and negative values indicating negative correlations.

C. Analysis Granularities

1) Granularity of Maintenance Work: Related work often considers software system versions that are not further developed but only maintained to facilitate identifying maintenance work. As systems in industry are often simultaneously maintained and further developed, we think it is crucial to also evaluate metrics in such a setting. As we can automatically distinguish between development activities and maintenance activities (cf. Section II-A), we can extract maintenance activities from the development histories of such systems.

Related work furthermore often aggregates maintenance activities across multiple months [8] or years [28], [31] into one data point. Thus, they calculate the code metrics at the beginning of each interval and merge all performed changes. This means, all activities during this time are combined and collectively related to the state of the system before the first change. We advocate this aggregation as problematic since maintenance activities are likely to be influenced by the changes made to the system during this time span. Thus, we prefer a task-oriented approach similar to Sjøberg et al. [9]: For each task, we consider the immediately preceding system state and only changes directly related to that task.

We utilize issues in the used issue tracker system to identify maintenance-related tasks (cf. Section II-A). This allows two possible interpretations of which code changes to consider. First, every commit linked to this issue represents a maintenance activity performed on the system. Therefore, we analyze the correlations between code attributes and the effort of each maintenance-related commit. Second, we can refer to the pull request that closed the issue in the issue tracking system. All

[2]https://github.com/ls1intum/Artemis
[3]https://github.com/ReactiveX/RxJava
[4]https://github.com/spring-projects/spring-framework

[5]https://www.cqse.eu/en/teamscale/overview/
[6]https://github.com/mauricioaniche/ck/

commits related to this pull request are aggregated into a single data point by summing up the RLOC values for duplicated files. We then calculate the inter-class metrics on the system state prior to the first commit of the respective issue, precisely the parent state of its first commit. This results in fewer data points for the analysis, but arguably represents the concept of a maintainability task best.

2) Time Granularity: For the commit-based and the issue-based analysis, we perform analysis over all data points per project. Additionally, we investigate whether the correlation between attribute metrics and maintenance effort is consistent for different time intervals throughout the maintenance process. The latter is motivated by the long period over which our data was collected. As we consider the complete life span of the study systems, the context is not constant and evolving over time. As a consequence, data drift is possible and may block the detection of important relations if all available data is considered. To account for this phenomenon, we split the available data into disjoint time intervals based on their date and ran an analysis separately for each interval. A good method to test all possible time intervals would be a sliding window algorithm with different window sizes to group data commits or issues based on their creation date for further analysis. However, as this kind of analysis would be very expensive in time and resources, we grouped the commits and issues based on their date per quarter of the year and analyzed the resulting groups independently. This allows to detect whether relationships are relatively consistent over time. In summary, we perform $2 \times 2 \times 2$ analyses for each study project, i.e. $\{\text{commit}; \text{issue}\} \times \{\text{unrestricted}; \text{time-restricted}\} \times \{\text{corrective}; \text{adaptive/perfective}\}$.

V. Commit-Based Analysis

In this section, we report the results of the analyses over all maintenance-related commits, separately for each of the software systems and maintenance types. Our supplemental material [32] reports the results in detail.

A. Complete Project History

The results for the three projects *Artemis*, *RxJava*, and *Spring-Framework* are shown from top to bottom, in Figure 1. Scatter plots show the magnitude of the Spearman correlation coefficient from -0.5 to +0.3 on the Y axis against each of the static code metrics on the X axis. For each metric, a dot represents the coefficient for the corrective maintenance type and a cross the coefficient for the adaptive/perfective maintenance type.

For *Artemis*, the highest observed correlation coefficients were found for the coupling metric ECM, with ≈0.13 for the corrective maintenance type and even lower with ≈0.08 for the coupling metric ECV for the adaptive/perfective type. The highest observed correlation coefficients for *RxJava* were found for the corrective maintenance type for the coupling metric ECV with ≈0.23 and for the adaptive/perfective type with ≈0.12. All cloning metrics show lower correlation coefficients than the coupling and inheritance metrics and

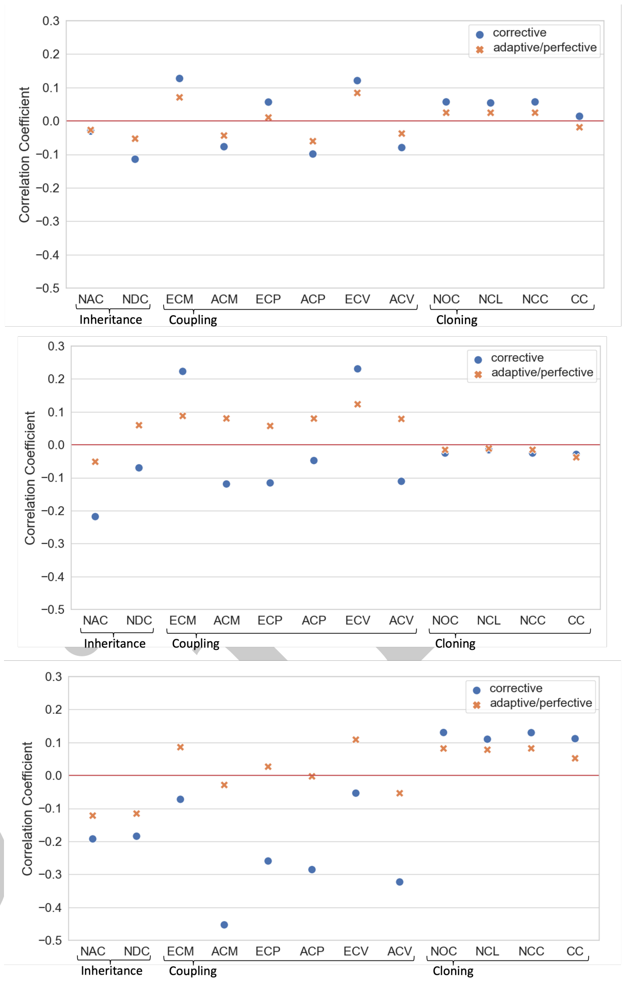

Fig. 1. Results for the commit-based analysis over the whole project history. From top to bottom for the projects *Artemis*, *RxJava*, and *Spring-Framework*.

show hardly any difference between the corrective and the adaptive/perfective maintenance type. For *Spring-Framework*, the highest observed correlation coefficient for the corrective maintenance type is higher than for *Artemis* and *RxJava*: the coupling metric ACM with ≈-0.45. For the adaptive/perfective maintenance type, the highest value is ≈-0.12 for the inheritance metric NAC.

B. Time-Restricted Analysis

In addition to the analysis over all commits, we investigated correlations between maintenance effort and code attributes in limited time intervals of three months. The results for the three projects *Artemis*, *RxJava*, and *Spring-Framework* are shown from top to bottom for the inheritance and coupling metrics in Figure 2 and from left to right for the cloning metrics in Figure 3. The box plots show the correlation coefficients on the Y axis from -1.2 to +1.2 for each of the inheritance and cloning metrics and from -0.6 to +0.8 for the cloning metrics. In all diagrams, for each metric, the box on the left represents the data for the corrective maintenance type, and the box on the right the data for the adaptive/corrective maintenance type. The variances between the correlation coefficients of the different time intervals are reported for each of the systems in Table III.

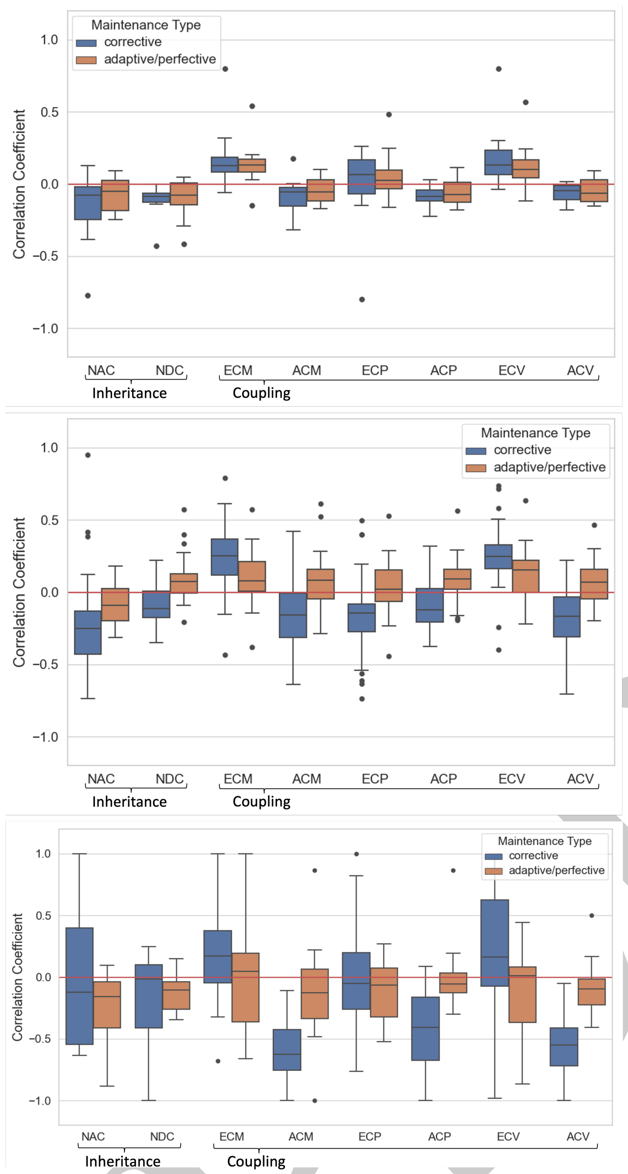

Fig. 2. Results for the time-restricted commit-based analysis of the inheritance and coupling metrics. From top to bottom for the projects *Artemis*, *RxJava*, and *Spring-Framework*.

The box plots for *Artemis* show that the correlation values for all inheritance and coupling metrics were less than 0.4 and greater than -0.4 for each of the time intervals and maintenance types apart from a few outliers which show correlation coefficients up to ±0.8. However, the median values for the coupling and inheritance metrics only lie between -0.13 and 0.13. For all cloning metrics, the results range between -0.2 and 0.3 and have no outliers. The results show positive as well as negative correlation values for each of the metrics and maintenance types. However, for both types, the median values are always either positive or negative for each of the metrics.

For *RxJava*, apart from a few outliers, the correlation coefficients for the inheritance and coupling metrics range from ≈-0.75 to ≈0.61 for the corrective and from ≈-0.29 to ≈0.4 for the adaptive/perfective maintenance type. However, the median values only lie between -0.25 and 0.25 for the corrective and ≈-0.15 and 0.2 for the adaptive/perfective maintenance type. The coefficients for the cloning metrics range between -0.4 and 0.6 including outliers. However, their median values only range from -0.05 to 0.12. Multiple metrics show a positive median for the adaptive/perfective and a negative median for the corrective maintenance type, namely NDC, ACM, ECP, ACP, and ACV.

For *Spring-Framework*, the results of the inheritance and coupling metrics range from -1.0 to 1.0 for the corrective maintenance type. Here, all metrics either have a minimum value of -1.0 or a maximum value of 1.0. Three coupling metrics show significantly higher medians for the corrective maintenance type than the other metrics, namely ACM with ≈-0.62, ACP with ≈-0.41, and ACV with ≈-0.55. The median values range for the adaptive/perfective maintenance type from ≈-0.2 to ≈0.1. For the cloning metrics, the results for the corrective maintenance type range from ≈-0.55 to ≈0.8. However, for the adaptive/perfective maintenance type, the results only range from ≈-0.35 to ≈0.2. The median values only range from 0 to ≈0.25 for both types.

VI. Issue-Based Analysis

As explained in Section IV-C1, we also investigate the relationship between software maintainability and metrics on the granularity of issues. Hence, we aggregate all commits associated with a specific issue into one data point. As the coupling and inheritance metrics consistently showed more promising results in the commit-based analysis than the cloning metrics, we concentrate on them for this analysis phase. See the supplemental material [32] for detailed results.

A. Complete Project History

For the analysis over the complete project history, we performed correlation analysis collectively over all issue-based data points for each of the maintenance types. The results for the three projects *Artemis*, *RxJava*, and *Spring-Framework* are displayed in the three diagrams of Figure 4. Scatter plots show the magnitude of the Spearman correlation coefficient from -0.4 to +0.2 on the Y axis against each of the inheritance and coupling metrics on the X axis. For each metric, a dot represents the coefficient for the corrective maintenance type and a cross for the coefficient for the adaptive/perfective maintenance type.

For *Artemis*, the correlation coefficients for both maintenance types range from ≈-0.14 to ≈0.03. The resulting values for corrective and adaptive/perfective maintenance are always very similar for each of the metrics with a discrepancy of at most 0.05. For *RxJava*, the correlation coefficients are more spread between the corrective and the adaptive/perfective maintenance type than for the *Artemis* project. They range from ≈-0.33 to ≈-0.02 for the corrective and from ≈-0.18

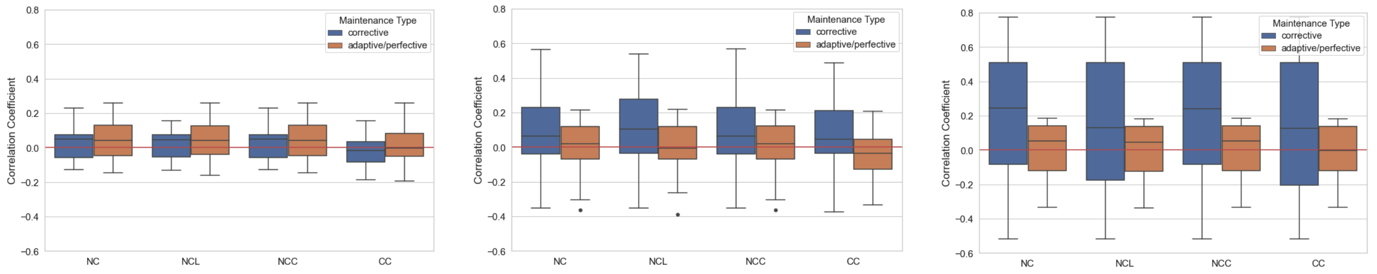

Fig. 3. Results for the time-restricted commit-based analysis of the cloning metrics. From left to right, the diagrams are for the projects *Artemis*, *RxJava*, and *Spring-Framework*

TABLE III
ROUNDED VARIANCE OF THE CORRELATION COEFFICIENTS OF BOTH TIME-RESTRICTED ANALYSES, FOR THE PROJECTS *Artemis*, *RxJava*, AND *Spring-Framework* IN NUMERICAL ORDER.

Project	Commit-Based												Issue-Based							
	Inheritance		**Coupling**						**Cloning**				**Inheritance**		**Coupling**					
	NAC	NDC	ECM	ACM	ECP	ACP	ECV	ACV	NDC	NCL	NCC	CC	NAC	NDC	ECM	ACM	ECP	ACP	ECV	ACV
1: corr	0.05	0.01	0.04	0.01	0.07	0.00	0.04	0.00	0.14	0.14	0.14	0.14	0.05	0.02	0.06	0.02	0.07	0.00	0.07	0.00
1: a/p	0.01	0.02	0.02	0.00	0.02	0.00	0.02	0.00	0.01	0.01	0.01	0.01	0.00	0.01	0.02	0.00	0.02	0.01	0.02	0.00
2: corr	0.12	0.02	0.06	0.07	0.09	0.03	0.06	0.06	0.05	0.05	0.05	0.05	0.07	0.02	0.09	0.07	0.06	0.03	0.08	0.06
2: a/p	0.02	0.03	0.03	0.04	0.03	0.03	0.03	0.02	0.02	0.02	0.02	0.02	0.04	0.02	0.03	0.04	0.04	0.03	0.03	0.04
3: corr	0.30	0.18	0.22	0.07	0.26	0.13	0.31	0.08	0.19	0.20	0.19	0.20	0.16	0.25	0.27	0.26	0.09	0.38	0.09	0.20
3: a/p	0.07	0.02	0.18	0.16	0.07	0.08	0.12	0.05	0.16	0.16	0.16	0.16	0.04	0.03	0.07	0.12	0.05	0.08	0.10	0.05

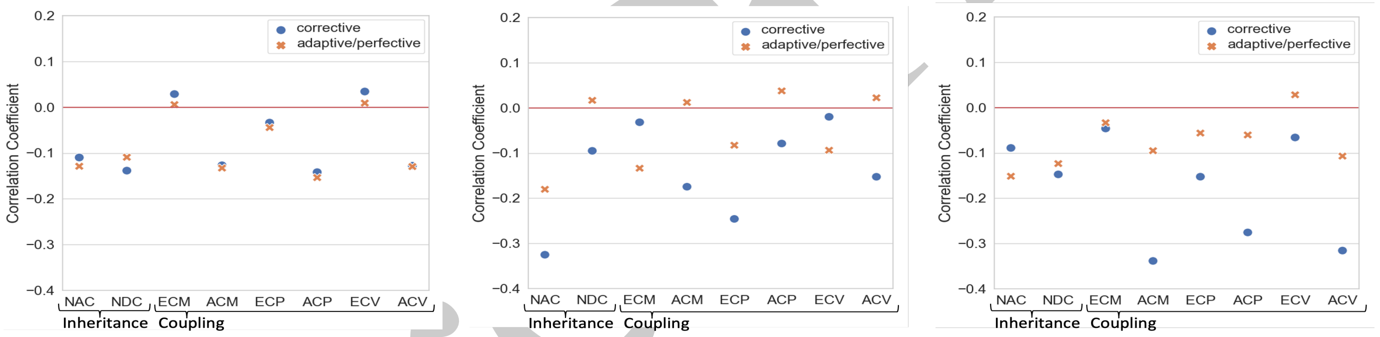

Fig. 4. Results for the issue-based analysis over the whole project history, for the inheritance and coupling metrics. From left to right, the diagrams are for the projects *Artemis*, *RxJava*, and *Spring-Framework*.

to ≈0.04 for the adaptive/perfective maintenance type. For *Spring-Framework*, the correlation coefficients range from ≈-0.34 to ≈-0.04 for the corrective and from ≈-0.15 to ≈0.03 for the adaptive/perfective maintenance type. The correlation coefficients for both maintenance types are close together for the NAC, NDC, and ECM metric with a discrepancy of at most 0.06. However, for the other metrics, the correlation coefficients for the corrective maintenance type are significantly higher than for the adaptive/perfective.

B. Time-Restricted Analysis

As explained in Section IV-C2, we examined whether there are correlations between maintenance effort and software metrics when analyzing specific periods of time separately, i.e. per quarter of the year. The results for the three projects *Artemis*, *RxJava*, and *Spring-Framework* are shown through box plots in the three diagrams in Figure 5. The box plots show the correlation coefficients on the Y axis from -1.2 to +1.2 for each of the code metrics on the X axis. For each metric, the box on the left represents the data for the corrective maintenance type and the box on the right the data for the adaptive/corrective maintenance type. The variances between the correlation coefficients of the different time intervals are reported for each of the systems in Table III.

For *Artemis*, the box plots show that the correlation values were less than 0.5 and greater than -0.5 for each of the time intervals and the maintenance types apart from a few outliers. The medians of all metrics range from ≈-0.17 to ≈0.07.

For *RxJava*, the correlation coefficients for the corrective maintenance type range from -0.7 to 0.6. For the adaptive/perfective type, the correlation coefficients range from

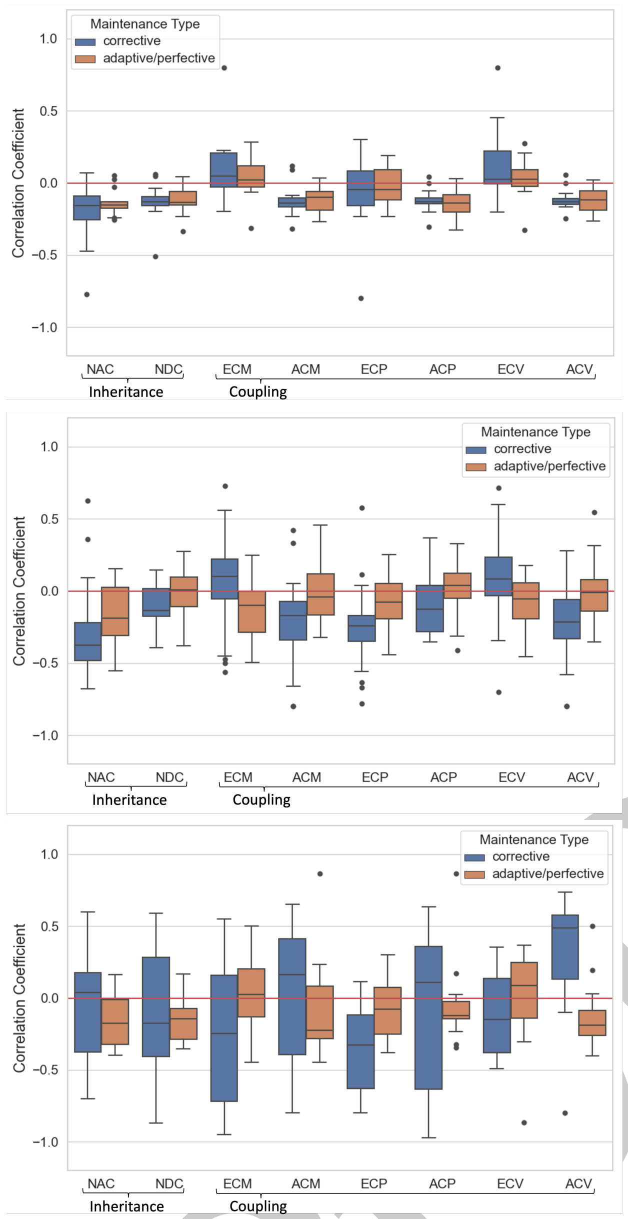

Fig. 5. Results for the time-restricted issue-based analysis of the coupling metrics. From top to bottom, the diagrams are for the projects *Artemis*, *RxJava*, and *Spring-Framework*.

≈-0.5 to ≈0.48 apart from one outlier. The median values for the corrective type range from -0.38 to 0.26. However, for the adaptive/perfective maintenance type, they only range from ≈-0.2 to ≈0.02 for all coupling and inheritance metrics.

For *Spring-Framework*, the correlation coefficients for the corrective maintenance type range from ≈-1 to ≈0.75. However, for the adaptive/perfective type, the correlation coefficients only range from ≈-0.45 to ≈0.37. The median values range for most coupling and inheritance metrics from ≈-0.38 to ≈0.2. However, for the ACV metric and the corrective maintenance type, the median value is ≈0.5.

VII. Discussion

A. Interpretation of Results

a) Consistent correlation for different projects: The analyses reveal differences in the correlation coefficients between the three projects for each of the metrics. For instance, the coupling metric ACM shows a correlation coefficient of -0.45 for the corrective maintenance type for *Spring-Framework* in the commit-based analysis over the complete project history. However, for *RxJava*, it only shows a correlation coefficient of -0.12 and for *Artemis* even less with a coefficient of -0.08. Although we interpret none of these coefficients as high correlation, the results still reveal an interesting aspect: If a metric shows a high correlation with maintenance effort in one project, and therefore has the capability to be a good measure for maintainability, this cannot be transferred to all other projects. Instead, the metrics need to be evaluated individually for each project they are used in.

b) Consistent correlation for different maintenance activities: In all analysis steps, for most projects, the metrics show higher correlations for the corrective maintenance type than for the adaptive/perfective. However, for *Artemis*, both maintenance types showed similar correlation coefficients for each of the metrics whereas *RxJava* and *Spring-Framework* showed larger differences between the two maintenance types. Additionally, for Artemis, the correlation coefficients for the two maintenance types are consistently either both positive or both negative except for one outlier, the cloning metric CC. Whereas for *RxJava* and *Spring-Framework*, multiple metrics show a positive correlation for one maintenance type and a negative correlation for the other. E.g., for *RxJava*, the inheritance metric NDC has a correlation coefficient of -0.07 for the corrective maintenance type and of +0.06 for the adaptive/perfective maintenance type in the commit-based analysis over the complete project history. Based on these results, broad inheritance, high coupling, or high cloning can either have a similar effect on maintainability for both maintenance types or reduce the effort for one maintenance type and increase the effort for the other.

c) Consistent correlation over time: Surprisingly, in all projects, most metrics show positive as well as negative correlations with maintenance effort in different time intervals. For instance, the coupling metric ECV shows correlation coefficients from -1.00 to +1.00 within the time-restricted commit-based analysis for *Spring-Framework*. Hence, based on these results, not only the strength in the correlation changes between time intervals but also whether high coupling, inheritance width, or cloning has a positive or negative impact on the effort of maintenance activities.

d) Notable Observations: The metrics show inconsistencies between the three projects, the two maintenance activity types, and time intervals. Most metrics show weak general correlations with maintenance effort with coefficients up to ±0.3 for most projects. Throughout the commit- and issue-based aggregation, we observe some consistencies within projects. However, the strength of the correlations varies.

E.g., for *Spring-Framework*, the coupling metric ACM has a correlation coefficient of -0.45 for the corrective maintenance type in the commit-based analysis over the complete project history, but a coefficient of -0.34 in the issue-based analysis.

As we did not expect this high variation between time intervals and maintenance types, we also investigated the LOC metric for comparison. This metric is considered the most promising measure for maintainability in related work [9]. However, the results for the LOC metric corroborate the results for the inter-class metrics: We cannot validate a general correlation. Instead, strong correlations only hold for specific projects and, within these projects, only for specific time intervals. Over the whole project history, however, the metric only shows weak correlations with maintenance effort.

B. Threats to Validity

1) Conclusion Validity: Although Spearman's rank correlation coefficient is a common correlation measurement in this research domain [6], [9], [33], it only detects monotonic relations. We acknowledge that other techniques might reveal different results. Saarimäki et al. [29] state that such analyses, in general, should consider the temporal context of the data. Unfortunately, their approach is not yet operationalized. So far, we did not investigate combinations of metrics, e.g. using machine learning [7], [31]. One of the first steps in building machine-learned models is to identify adequate feature candidates. In our study, we found only inconsistent correlations with the given maintainability label, with correlation coefficients oscillating between positive and negative values. Based on these observations, we question the usefulness of the studied inter-class metrics as input for machine learning.

2) Internal Validity: Though widely used and accepted in this area of research [9], [16], [28], RLOC only provides an approximation of maintenance effort. Although it is found to correlate with the time needed to perform maintenance work [17], it is used under the assumption that every changed line approximately needs the same effort. Still, we chose this surrogate measurement, as it is automatically extractable and allows analyzing systems with multiple years of maintenance. However, a different variable could have shown different correlation results. Furthermore, we referred to tags of the issues to classify and identify maintenance work. We verified for a randomly selected subset that the assigned tag matches the issue description. Still, we cannot guarantee the correctness and completeness of the assigned tags.

3) External Validity: First, the collected maintenance and metric data greatly depend on the chosen software systems. Therefore, other study projects might have shown different results and it is unclear, whether our results generalize. However, it is the very point of this work to show, that insights from one project and time interval hardly translate to other contexts. Second, we have not covered all possibilities in software systems or programming languages, as all systems are open-source and mainly written in Java. Therefore, our results are not generalizable to all types of software projects. Third, we have not exhausted all possible system sizes. However, our software systems were significantly larger than in related work [8], [9], [28]. Fourth, since we used the tags of issues to identify maintenance work, we had to limit our selection of systems to systems where such tags are used. Therefore, there could exist a bias in our data as these systems might be developed and maintained in a better, more structured way than average systems.

VIII. Related Work

Several studies investigated the relationship between software attribute metrics and maintainability. One discriminating factor is their measurement of maintainability. Sjøberg et al. [9] examined the Maintainability Index, a class cohesion metric, two size metrics, one coupling metric, and the Depth of Inheritance Tree (DIT) metric on their abilities to measure maintainability. To quantify the maintenance effort, they recorded the time developers spent on each file while performing dedicated maintenance tasks and calculated the number of changed lines. Apart from size and class cohesion, the metrics are very inconsistent between different software systems. Only LOC showed a perfect correlation with maintenance effort in hours, but only a correlation coefficient of 0.4 with RLOC. Dagpinar and Jahnke [8] used the frequency of maintenance activities to quantify maintenance effort and distinguished between different types of maintenance tasks. However, they categorized the maintenance tasks manually based on commit messages. Their results indicate that size and efferent coupling metrics are good measures for the maintainability of software classes, while inheritance, cohesion, and indirect and afferent coupling metrics are not. Al Dallal [28] examined 19 metrics for size, cohesion, and coupling. He used the number of RLOC and the number of revisions to quantify the maintenance effort. The results indicate that classes with smaller size, lower coupling, and higher cohesion have higher maintainability.

Several studies focused explicitly on the influence of inheritance depth on maintainability [10], [11], [15]. These studies share several characteristics: they perform controlled experiments with students, use dedicated systems and specifically designed maintenance tasks, and use the DIT metric to express inheritance relationships. In contrast to these studies, we considered real software systems, actual past maintenance activities, and two improved inheritance metrics.

Furthermore, there are several approaches for predicting maintainability using machine learning. Most approaches are based on a data set created by Li and Henry [31] in 1993, which contains a small number of code files written in Classic-Ada. It is questionable whether the results obtained on this data hold for systems in modern programming languages. Other works refer to manually-labeled data [14]. Using this data, size metrics and the cognitive complexity metric were found to offer the highest predictive power [7].

In contrast to related work, we compare the measuring capabilities of metrics on two different aggregation levels: single commits and complete maintenance tasks. Furthermore, the considered systems are significantly larger than those analyzed by related work. Additionally, we distinguish maintenance types, which is rarely done in related work.

IX. Conclusion

With this study, we evaluate 11 inter-class metrics and the LOC metric for their capabilities of assessing the maintainability of software systems. We leverage modern repository mining techniques and analyze the maintenance history of three large open-source software systems with 1,095 to 7,217 Java files. Unlike most related work, we automatically identify maintenance activities and distinguish between different types of maintenance. Furthermore, we perform a more in-depth analysis than related work to examine the influence of the considered time interval, maintenance type, different aggregation levels of maintenance work, and the project specificity of the results.

Our study reveals that none of the metrics consistently correlates with maintenance effort. Instead, the relationship varies between the project, granularity of the maintenance activity, type of maintenance, and considered time interval. The variation between different projects has the consequence that metrics that predict maintainability well in one project do not necessarily predict maintainability in other projects. However, the variation over time has more severe impacts. Some metrics show high correlations with maintenance effort with correlation values up to ± 1.00. However, these high correlation coefficients only hold for specific projects, and within these projects, only for limited time spans. Each metric shows positive as well as negative correlations with maintenance effort for different time intervals and most of the systems. Based on these results, not only the strength in the correlation changes between time intervals but also whether high coupling, inheritance width, cloning, or size has a positive or negative impact on the effort of maintenance activities. This means that metrics that are good predictors of maintainability within one project at one point in time, do not necessarily stay good predictors. One implication of these findings is that machine learning models trained on these metrics or other metric aggregations also need project-specific and time-specific validation.

Additionally, our results suggest that high coupling, inheritance width, cloning, and size can influence the effort for different maintenance activities differently. For two of the three projects, multiple metrics even show a positive correlation for one maintenance type and a negative correlation for the other.

We, therefore, conclude that the evaluated code metrics should not be used to assess the maintainability of software systems without context-specific validation. If metrics are to be used, it is necessary to regularly verify that they still correlate with maintenance effort in that project.

References

[1] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[3] M. Muñoz Barón, M. Wyrich, and S. Wagner, "An empirical validation of cognitive complexity as a measure of source code understandability," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ACM, 2020.

[4] M. Nilson, V. Antinyan, and L. Gren, "Do internal software quality tools measure validated metrics?," in *International Conference on Product-Focused Software Process Improvement (PROFES)*, pp. 637–648, Springer, 2019.

[5] T. Sharma and D. Spinellis, "Do We Need Improved Code Quality Metrics?," *arXiv:2012.12324 [cs]*, Dec. 2020. arXiv: 2012.12324.

[6] T. Zimmermann, N. Nagappan, and A. Zeller, "Predicting bugs from history," in *Software evolution*, pp. 69–88, Springer, 2008.

[7] M. Schnappinger, A. Fietzke, and A. Pretschner, "Human-level ordinal maintainability prediction based on static code metrics," in *Evaluation and Assessment in Software Engineering*, EASE 2021, (New York, NY, USA), pp. 160–169, Association for Computing Machinery, 2021.

[8] M. Dagpinar and J. Jahnke, "Predicting maintainability with object-oriented metrics -an empirical comparison," in *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, (Victoria, BC, Canada), pp. 155–164, IEEE, 2003.

[9] D. I. K. Sjoberg, B. Anda, and A. Mockus, "Questioning software maintenance metrics: A comparative case study," in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 107–110, Sept. 2012.

[10] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "Evaluating inheritance depth on the maintainability of object-oriented software," *Empirical Software Engineering*, vol. 1, pp. 109–132, Jan. 1996.

[11] M. Cartwright and M. Shepperd, "An empirical view of inheritance," *Information and Software Technology*, vol. 40, pp. 795–799, Dec. 1998.

[12] L. Prechelt, B. Unger, M. Philippsen, and W. Tichy, "A controlled experiment on inheritance depth as a cost factor for code maintenance," *Journal of Systems and Software*, vol. 65, pp. 115–126, Feb. 2003.

[13] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the International Conference on Software Engineering*, pp. 492–497, 1976.

[14] M. Schnappinger, A. Fietzke, and A. Pretschner, "Defining a Software Maintainability Dataset: Collecting, Aggregating and Analysing Expert Evaluations of Software Maintainability," in *IEEE International Conference on Software Maintenance and Evolution*, pp. 278–289, 2020.

[15] R. Harrison, S. Counsell, and R. Nithi, "Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems," *Journal of Systems and Software*, vol. 52, pp. 173–179, June 2000.

[16] W. Albattah, "An Empirical Investigation of the Correlation between Package-Level Cohesion and Maintenance Effort," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 3, 2017.

[17] J. Hayes, S. Patel, and L. Zhao, "A metrics-based software maintenance effort model," in *Eighth European Conference on Software Maintenance and Reengineering, CSMR 2004*, pp. 254–258, Mar. 2004.

[18] W. Li, "Another metric suite for object-oriented programming," *Journal of Systems and Software*, vol. 44, pp. 155–162, Dec. 1998.

[19] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, Dec. 1990.

[20] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, pp. 91–121, Jan. 1999.

[21] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[22] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software*, vol. 86, pp. 2639–2653, Oct. 2013.

[23] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum, 2007.

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

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

[26] C. J. Kapser and M. W. Godfrey, """Cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645–692, Dec. 2008.

[27] "How Maintainable is Your Code? | Teamscale Documentation."

[28] J. Al Dallal, "Object-oriented class maintainability prediction using internal quality attributes," *Information and Software Technology*, vol. 55, pp. 2028–2048, Nov. 2013.

[29] N. Saarimäki, S. Moreschini, F. Lomio, R. Penaloza, and V. Lenarduzzi, "Towards a robust approach to analyze time-dependent data in software engineering," 2022.

[30] *Spearman Rank Correlation Coefficient*. Springer New York, 2008.

[31] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, pp. 111–122, Nov. 1993.

[32] L. Gregor, M. Schnappinger, and A. Pretschner, "Revisiting Inter-Class Maintainability Indicators - Supplemental Material." https://doi.org/10.6084/m9.figshare.21571905.v1, Mar. 2023.

[33] R. Harrison, S. Counsell, and R. Nithi, "Coupling metrics for object-oriented design," in *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No.98TB100262)*, pp. 150–157, Nov. 1998.