Defining a Software Maintainability Dataset: Collecting, Aggregating and Analysing Expert Evaluations of Software Maintainability

Markus Schnappinger
Technical University of Munich
Munich, Germany
markus.schnappinger@tum.de

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

Alexander Pretschner
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

***Abstract*—Before *controlling* the quality of software systems, we need to *assess* it. In the case of maintainability, this often happens with manual expert reviews. Current automatic approaches have received criticism because their results often do not reflect the opinion of experts or are biased towards a small group of experts. We use the judgments of a significantly larger expert group to create a robust maintainability dataset. In a large scale survey, 70 professionals assessed code from 9 open and closed source Java projects with a combined size of 1.4 million source lines of code. The assessment covers an overall judgment as well as an assessment of several subdimensions of maintainability. Among these subdimensions, we present evidence that understandability is valued the most by the experts. Our analysis also reveals that disagreement between evaluators occurs frequently. Significant dissent was detected in 17% of the cases. To overcome these differences, we present a method to determine a consensus, i.e. the most probable true label. The resulting dataset contains the consensus of the experts for more than 500 Java classes. This corpus can be used to learn precise and practical classifiers for software maintainability.**

***Index Terms*—Software Maintenance, Software Quality, Machine Learning, Software Measurement**

I. Introduction

A. Background and Motivation

The quality of a software system is an important determinant of its economic profitability. Especially maintenance costs significantly contribute to the overall costs of a software system [1]. Consequently, companies try to maintain their systems as cost-efficiently as possible. Continuous quality control helps in identifying potentials for improvements early [2]. Expert reviews, for example, are a very precise evaluation method and can even be tailored to specific business goals [3]. Automatic approaches, on the other hand, are objective and fast. While the key problem with expert reviews is their time-consuming and expensive nature [4], [5], automated tools are often imprecise and the results need to be interpreted [6]. Promising results have been achieved using supervised machine learning to predict the maintenance effort of a program. To evaluate the performance of the different prediction approaches, a variety of maintainability datasets are used. While some define maintainability as the number of observed changes [7], others use formal metric definitions [8]. Just as maintainability is a subcategory of quality, maintainability can also be further subdivided [9], [10]. However, few authors have considered this distinction and define precisely which of these aspects is predicted in the study. A positive example is the work of Buse and Weimer [11], [12] and then Posnett et al. [13], who developed a model to explicitly capture readability.

To automate expert reviews, the most promising approach is arguably to use expert judgment as the ground truth. Garvin was among the first to state that quality is a viewpoint dependent attribute [14]. Therefore, it is crucial to investigate and report how the experts form their opinion. Ideally, a consensus of several experts is found as proposed by Rosqvist et al. [15]. Though the ISO/IEC standard deliberately defines quality vaguely and requires interpretation and tailoring [16], one must understand the causal factors leading to a qualitative statement. This is necessary because automated approaches rely on well-defined labels. In the area of maintainability prediction, there is little published data on which aspects the experts considered in their evaluations and no investigation of the disagreement between experts. However, the prediction models developed by Pizzi et al. [17] and Hegedűs et al. [18] are based on a dataset created by only one expert. Another study [19] uses data labeled by three experts, but omits the exact labeling procedure. The data of all three studies are not publicly available for replication studies.

B. Contribution

In this paper, we present a large scale study that creates a robust software maintainability dataset based on expert evaluations. In total, 70 software analysts, researchers, and developers working with e.g. Facebook, Oracle, or BMW contributed around 2,000 manual assessments. This dataset can be used to develop automated prediction tools. In contrast to most other work using expert assessments, we analyse in depth which aspects the experts had taken into account during the assessment. In our study, we collected evaluations of overall maintainability and several subcategories thereof. Eventually, the created dataset contains labels for 519 Java classes. The findings from this study make several contributions to the current literature: (1) we introduce a dataset for

2576-3148/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSME46990.2020.00035

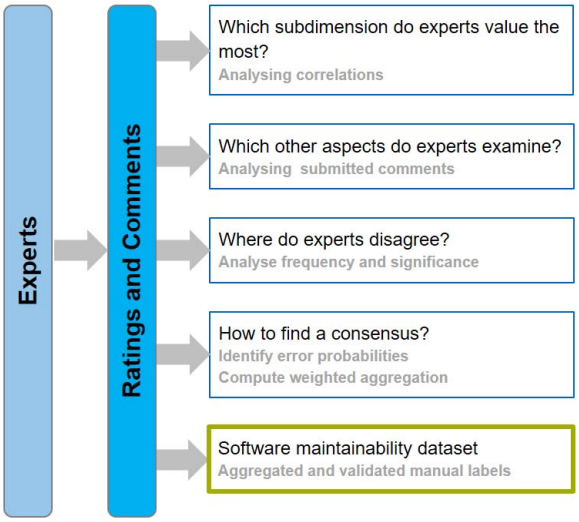

Fig. 1. Research Questions and Analyses.

future use in the maintainability research community; (2) an adaption of an algorithm for automatically finding consensus between experts; (3) several insights into expert judgments of maintainability, including the diversity of opinions and weighting of subcharacteristics; and (4) empirical evidence on what makes code less maintainable, including commented-out code, hardcoded values and quality of the comments.

C. Research Questions

The primary goal of this research is to create a large and robust software maintainability dataset. To maximize the potential of the dataset for future research, our study answeres the following research questions:

It is known that maintainability is determined by several subaspects. We hypothesize that these subaspects all contribute to the overall evaluation of maintainability, but they do not contribute equally. Thus, we pose our first research question: Which subdimension of maintainability do experts value the most? To put the results of this question into context, we extend the focus of the data: Which other aspects do experts consider to assess the quality of code?

Next, we investigate dissent and consensus between experts. Based on the fact that quality is inherently subjective, we put forward the hypothesis that expert opinions may differ. Consequently, we investigate: In which cases do experts disagree and how severe is the dissent? Assuming that experts may disagree, the need to find a consensus between them emerges. This motivates the next question: How can we automatically aggregate the opinions of several experts, not knowing upfront which expert is right? Eventually, our insights are used to construct a useful and robust software maintainability dataset.

II. Study Design

This section introduces the study objects, i.e. the analysed source code, the recruited experts, and which labels they will assign. As Fig. 1 shows, our research questions and analyses are based on the ratings and comments provided by experts. An insightful dataset is necessary to answer the research questions. This dataset should consist of transparent labels provided by qualified experts. Also, the systems should cover diverse domains to mitigate domain-specific bias. To foster the practical relevance, the systems should be written in a modern and often used programming language. Though open source and closed source software share common characteristics, they also differ in some aspects [20]. To achieve better generalizability of the results, the dataset should contain projects of both types.

A. Study Objects

We took code snippets from 9 different projects. These projects are both open source projects and commercially developed products. All chosen projects are developed in Java. The sample covers a diverse range of domains and release dates. We believe this to reflect real-world software systems where quality reviews are applied. In a study by Ahmad and Laplante [21], Art of Illusion was found to be a very complex system. ArgoUML has previously been used in other qualitative studies concerned with internal software quality [22]–[24] as well. Hence, these systems seem appropriate choices with existing quality flaws. In contrast, JUnit 4 is a very popular open source framework for unit testing. We hypothesize this system will not yield many quality issues.

The commercial systems are developed by two different vendors and are taken from two separate software ecosystems. Therefore, we consider these 9 projects to portray a corpus with a high representativity. However, the study objects do not cover every possible domain and only the Java programming language. We decided to stick with one programming language only to avoid context switching during the labeling process. Java is considered one of the most popular programming languages and there is a high demand for Java in industry[1]. Hence, it is a reasonable choice. We selected the following projects as sources for the code snippets:

- Open source projects:
  - **ArgoUML**[2]**:** Tool to design, simulate and generate code from UML diagrams
  - **Art of Illusion**[3]**:** 3D Modeling and Rendering Studio
  - **Diary Management**[4]**:** Multi-user calendar tool
  - **JUnit 4**[5]**:** Testing framework for Java programs
  - **JSweet**[6]**:** Transpiler to convert Java code into Javascript or Typescript code
- Commercial projects, anonymized:
  - ***xApp*:** App used by insurance damage assessors
  - ***xBackend*:** Backend of an insurance system
  - ***xDispatch*:** Planning of personnel dispatch
  - ***xPrinting*:** Printing and layouting of documents

[1] https://cce.fortiss.org/trends/radar/compare?technologies=java
[2] https://github.com/argouml-tigris-org/argouml
[3] http://www.artofillusion.org/
[4] https://sourceforge.net/projects/diarymanagement/
[5] https://junit.org/junit4/
[6] http://www.jsweet.org/

TABLE I
INCLUDED SAMPLE PROJECTS

Project	Domain	Release	Size in SLOC	Files
ArgoUML	UML Models	2010	177 k	1,904
Art of Illusion	3D Modeling	2013	118 k	470
Diary Management	Calendar	2014	17 k	131
JSweet	Code Transpiler	2019	79 k	1,933
JUnit 4	Software Testing	2014	25 k	383
xApp	Insurance	2018	28 k	223
xBackend	Insurance	2018	82 k	637
xDispatch	Scheduling	2019	472 k	5,192
xPrinting	Printing	2019	435 k	4,841
In total:			**1.43 M**	**15,714**

In total, these systems consist of 15,714 Java files and account for 1.4 million source lines of code (SLOC), i.e. lines of code without comments and blank lines. Table I lists more details about the single projects including the release year. Since Java is a rapidly evolving programming language, we wanted to include code that does use newer features of Java as well as older code that sticks to basic features. Furthermore, we deliberately did not use the latest release for every project, e.g. ArgoUML. This enables future research to compare the maintainability, i.e. the maintenance effort predicted by the experts, to the actually required effort.

Each data point in our dataset represents one *.java* file, which - following the Java conventions - is equivalent to one programming class. Too short code snippets might not provide enough context for the analyst to reason about e.g. complexity or understandability. Analysing larger chunks of software such as packages, on the other hand, is very time-consuming and not suitable to collect many evaluations. A class-level analysis is a reasonable compromise between displaying context and the time needed to assess the data. Most static analysis tools also support class-level analysis, and this granularity is chosen by related work as well [6], [7], [19], [21], [25], [26].

B. Study Participants

The target group of our study are software quality analysts, researchers with a background in software quality, and software engineers that are involved with maintaining software. Some participants have up to 15 years of experience in quality assessments. In sum, 70 professionals participated. First, we invited selected experts to participate in the study. Second, we asked them to disseminate the study to interested and qualified colleagues. The survey was also promoted in relevant networks. The participants are affiliated with companies including Airbus, Audi, BMW, Boston Consulting Group, Celonis, cesdo Software Quality GmbH, CQSE GmbH, Facebook, fortiss, itestra GmbH, Kinexon GmbH, MaibornWolff GmbH, Munich Re, Oracle, and three universities. However, 7 participants did not want to share their affiliation.

C. Label Definition

The label represents the perceived quality of a code snippet. To be relatable, it should be as objective as possible. This is a non-trivial task given that the perception of quality may vary. We are meeting this challenge with two measures: finding the consensus of several opinions and limiting the possible perspectives under which quality is assessed. Consequently, we have to explicitly define which quality attributes the experts should focus on during the survey.

This study focuses on maintainability. Maintenance costs are the biggest contributor to the economic effectiveness of software systems [2], thus making maintainability a relevant research area. Also, expert assessments are particularly valuable here as maintainability evaluations cannot be obtained automatically. In the context of this work, we understand maintainability as the estimated future maintenance effort of a program class. This effort is influenced by several factors. During quality assessments, analysts investigate different aspects of quality. To give a final assessment, they negotiate with themselves a weighting of the observed dimensions. In our study, we reproduce this thought process. The study participant first evaluates the code with regard to various more fine-grained criteria before giving a final statement on maintainability. To record these ratings, a labeling platform was implemented. Section III-A describes it in detail.

Here, we try to capture the expert opinion as directly as possible. Consequently, we refrain from formal definitions and capture the concepts as the analysts perceive them. In their seminal book on survey design, Saris and Gallhofer point out this important distinction between concepts-by-postulation and concepts-by-intuition [27]. Asking an expert how they perceive the complexity of a code snippet aims at the intuition of the analyst, whereas measuring complexity using McCabe's cyclomatic complexity [28] is an example of the use of postulation. One way to elicit opinions about concepts-by-intuition is to pose statements and ask the participants whether they agree or disagree [27]. We advocate a four-part Likert-scale ranging from *strongly agree* to *strongly disagree*. From our point of view, binary values do not reflect the way experts perceive quality. In contrast, too many different values to choose from might overwhelm the experts and borders between the single categories will blur. Besides, a four-part scale deprives participants of the possibility of choosing a neutral position. Experts are forced to at least indicate a tendency. Some studies [29] suggest asking study participants how confident they are that their answer is correct. This is rendered obsolete by the scale chosen, as the experts can express their confidence with a clearer label on the four-part scale. In favour of a faster labeling process, we limited the survey to five dimensions: readability, understandability, complexity, adequate size, and overall maintainability. The more dimensions have to be considered, the more effort is needed for the evaluation. This impacts the amount of data that can be labeled in a given time. Besides, a more tedious and more complex labeling process might decrease participant motivation. It is unlikely that any set of questions can cover the entire spectrum of software maintainability. Therefore, we dedicate one question to the overall judgment. This means that all subaspects that have not been covered by an explicit

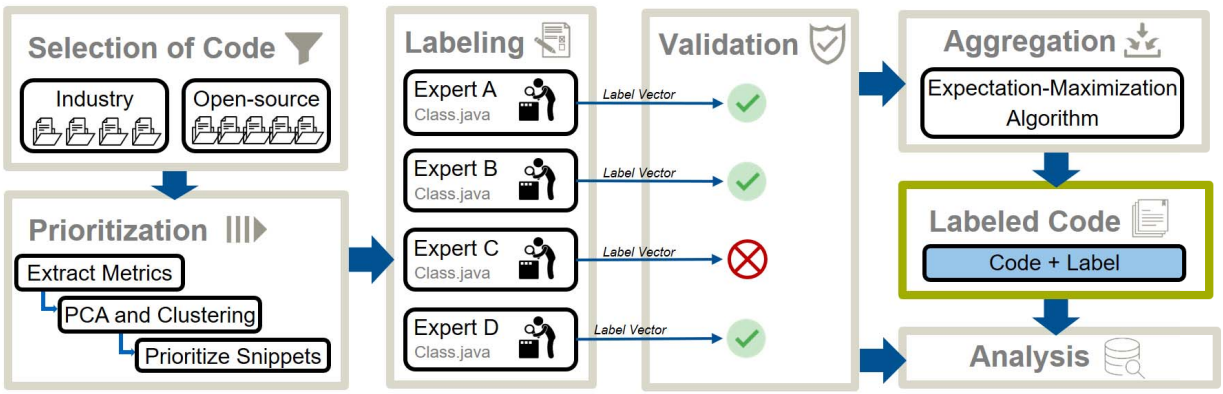

Fig. 2. Activities during the dataset creation: selection of code, prioritization of samples, labeling, validation, aggregation and analysis.

question can still influence an expert's overall evaluation. In the following, we explain the chosen dimensions and reasons to select them. One key activity in software maintenance is reading the existing source code [30]. Readability captures how easy it is to syntactically parse the code. It is concerned with e.g. indentation, line length, and identifier length. After reading the code, all maintenance activities require maintainers to comprehend the semantics of code they are going to adapt [31]. Therefore, we are interested in assessing the understandability of code. This attribute captures the effort to understand which concepts the code implements and to easily identify at which point a desired change must be made. This concept is known to affect maintainability [32]. Understandability and readability are related but need to be treated separately [13]. The distinction becomes clearer if we consider very short variable names of just one character. Those are easily readable, but might not yield enough information to comprehend their meaning. In the next dimension, we attempt to measure the complexity of a particular piece of code. This is especially interesting, since most other approaches like e.g. McCabe's cyclomatic complexity [28], try to formalize complexity. In contrast, we explicitly refrain from that and use the concept-by-intuition. The last considered criterion is the adequate size of the program class. One important aspect of maintainability is modularity. The concept of modularity as we usually understand it in software engineering cannot be transferred to class-level code snippets one to one. Still, it is partially applicable in the meaning of adequate sizing. In our study, participants can express whether they think the code should be split up into smaller snippets. This applies to both the size of the class itself as well as its e.g. extra-long methods. As explained above, the fifth label is the overall maintainability of the code snippet according to the personal intuition.

D. Availability of the Dataset

The interest in automating software quality evaluations is growing. Using publicly available data fosters the reproducibility of studies and enables fair comparisons. The results of this study, i.e. the analysed open source code and the corresponding labels, are shared in [33]. Please note that we only distribute the code of the non-proprietary systems. The archive also contains a summary of the dataset's threats to validity.

III. Data Collection

Before we can answer the research questions, we have to create a meaningful dataset. Such a dataset should consist of representative data points and reliable labels. One of the first things to consider is which projects to choose the samples from and which experts to recruit for the evaluations. We elaborated on that in Section II, whereas this section focuses on how exactly we collected the data. The interaction of all data collection activities is visualized in Fig. 2. The first step is to extract code snippets from the study projects and define in which order the snippets will be analysed. The study participants then evaluate the code and assign labels. Every code snippet is rated by several experts. Afterward, the collected labels are validated. Eventually, we apply an aggregation step. In this step, error probabilities for each participant are computed and their consensus is determined.

A. Labeling Platform

The labels are collected per an easy-to-use online tool. Since the study participants are volunteers, it is crucial to keep them motivated. The code evaluation platform provides a modern and intuitive frontend. No training period is needed. Every participant owns a password-protected account. This enables a sophisticated permission concept. The code of the commercial systems is restricted to employees of that company and to users owning explicit rights to inspect the code. Open source code can be rated by every participant. After logging in, a code snippet is presented to the user. To read the code as conveniently as possible, users can select their favourite syntax highlighting theme.

We ask the participants to rate each code snippet in five dimensions: readability, understandability, complexity, modularity, and overall maintainability. For every dimension, we post one statement and the analysts express their opinion on a four-part Likert-scale:

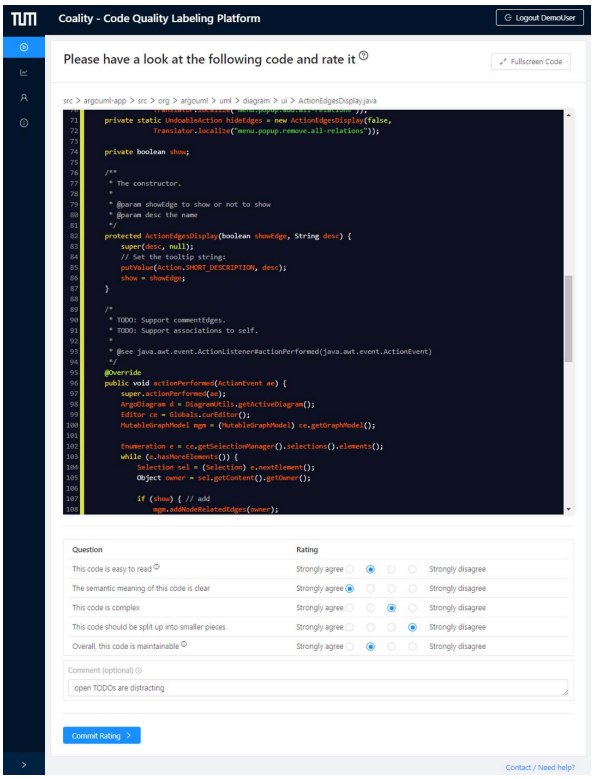

Fig. 3. Screenshot of the labeling platform showing code and questionnaire.

- *This code is easy to read*
- *The semantic meaning of this code is clear*
- *This code is complex*
- *This code should be split up into smaller pieces*
- *Overall, this code is maintainable*

Besides, the platform allows to add free text comments as well. Fig. 3 shows a screenshot of the survey tool. To foster participation and keep the users motivated, the study implements gamification elements. For example, confetti animations and motivational messages are displayed once the user has committed a certain amount of labels. Ambitious users can compare their performance with others on a public scoreboard. To follow data privacy rules, participants must explicitly opt-in to appear in this scoreboard.

B. Prioritization of Samples

We use static code metrics to guide the selection of code snippets. To make efficient use of the experts' time, the snippets are labeled in a specific order. First, the extracted metrics are used to cluster the code. Then, we prioritize them by iterating through these clusters. The next code snippet to be labeled is the snippet with the highest priority, from a codebase to which the user has access rights, and that did not yet receive enough valid ratings.

The combined size of the sample projects exceeds 1.4 million source lines of code and contains more than 15,000 files. The participation happens voluntarily, thus making it

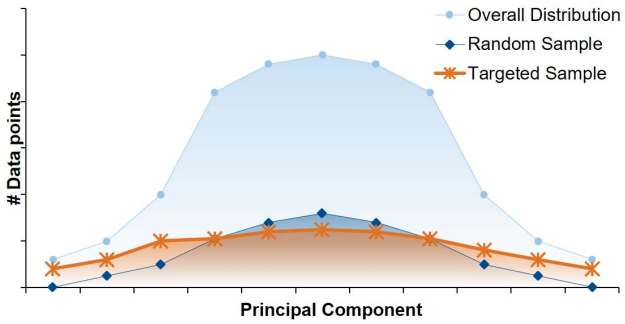

Fig. 4. The distribution of the targeted sample in comparison to a random sample.

illusional to aim for a dataset of 15,000 manually evaluated classes. Paying study participants is beyond the possibilities of this study due to the required expert knowledge. Since we cannot reason about the time the experts will dedicate to labeling, it is not possible to determine a feasible sample size apriori. Hence, we apply prioritization instead of sampling.

From our experience, quality reviews target to identify spots that are worth further investigation. A useful dataset must hence include such cases. However, focusing too much on suspicious data points would jeopardize the representative nature of the dataset. Fig. 4 illustrates sample distributions plotted towards an arbitrary dimension. The total distribution, which is approx. normally distributed, is visualized as the bell-shaped light blue curve. A random sample of, e.g., 25%, would probably correspond to the dark blue curve. In the diagram, one can see that this sample would hardly contain any edge cases, neither on the small nor on the large end of the axis. From quality assessments we know that classes with extraordinary values provide the most insights to the analysts. Therefore, we target a distribution that resembles the orange curve. This curve has two main characteristics: First, it contains significantly more edge cases than the random sample. Second, it still roughly resembles the overall distribution and does still contain many non-edge instances.

The theoretical foundation of our approach is as follows: Unsupervised machine learning, i.e. clustering, groups data points together that are more similar than others. The clustering algorithm uses static software metrics as characteristics to define the clusters. These include structural metrics concerned with nesting or size as well as more complex metrics like the number of code smells. A list of the used metrics and tools is published with the final dataset in [33]. Values are normalized and Principal Component Analysis avoids that clusters are distorted because of metrics measuring related properties. Then, *k-Means* is run on the principal components with k determined by the Expectation-Maximization Algorithm [34]. Some data points are more insightful than others because they somehow differ from the average sample. We hypothesize that those interesting cases can most probably be found in clusters with only a few other members. Consequently, we prioritize

TABLE II
ILLUSTRATION OF THE PRIORITIZATION ALGORITHM

Cluster	# files	Priorities assigned to files in this cluster	# labeled files	# labeled files if random
Cluster 0	10	1; 7; 21; 35;...	4	1
Cluster 1	46	2; 8; 22; 36;...	4	4
Cluster 2	55	3; 9-10; 23-24; 37-38;...	7	6
Cluster 3	68	4; 11-12; 25-26; 39-40;...	7	7
Cluster 4	109	5; 13-15; 27-29; 41-43;...	10	11
Cluster 5	182	6; 16-20; 30-34; 44-47;...	15	18

the data points starting with the smallest clusters. But, taking only the potentially interesting files would lead to a dataset that does not reflect the generality. Therefore, we only add a relative share of $max(1, 0.03 * sizeofcluster)$ points of each cluster to the prioritization queue. To ensure that at least one sample of every cluster is labeled, we perform a start-up round. Here, we label exactly one class from every cluster.

We illustrate the effect of this using the Art of Illusion sample project. There were six clusters identified in this project. One of the clusters contains only 10 files; we refer to this as *Cluster0*. Let us assume that the experts manage to label 10 percent of the files in this project, i.e. 47 files. If we chose randomly which files to label, the probability that at least one of the files from *Cluster0* is among the labeled files is $1 - \frac{\binom{47}{0}\binom{460}{47}}{\binom{470}{47}} = 65.5\%$. With our approach, the probability to have at least one file from *Cluster0* in the sample is 100%. In fact, there will be exactly four files from *Cluster0* in the sample. Table II illustrates the algorithm. In the start-up round, one class from every cluster is labeled, starting with the smallest cluster, i.e. *Cluster0*. In the next iteration, we select one sample from *Cluster0*, one from *Cluster1*, two from *Cluster3* and *Cluster4* resp., three from *Cluster4*, and five from *Cluster5*. This is performed iteratively until all files are prioritized. Assuming that 47 data points could be labeled, the final number of labeled data points per cluster is denoted in the third column of Table II. The right-most column shows how the distribution would look like on average had we used random prioritization. It becomes obvious that our approach shifts the focus from the large clusters to the smaller clusters, where the most interesting files are located. Simultaneously, the overall distribution of data points is respected.

C. Validation and Seriousness Checks

Every assessment is validated before it is included in our dataset. The target group of our study are professional software analysts, developers, and researchers. But since the labeling platform is publicly accessible, the motivation of a participant in general remains unknown and every participant needs to be treated agnostically. Analysing the data for inconsistencies can help to identify unserious participation [35]. In our study, we take the combination of quality dimensions into account and compare the values to known illicit combinations. E.g. if a participant states the code snippet was easily maintainable, while at the same time expresses it was neither readable nor understandable and highly complex, that rating is considered implausible. In that case, the plausibility score $p \in [0; 1]$ is set to 0, and 1 otherwise. The time needed to complete a task can be used to filter out noise by spammers or unserious contributors as well [36]–[38]. The labeling platform measures the time taken to create a rating. The number of characters an expert assesses per second arguably follows a normal distribution. We assume the faster an evaluation was finished, the more implausible it becomes. If a rating was created significantly faster than the average, i.e. more than two standard deviations faster than the average, this rating is discarded. If it has been created only slightly faster than the mean, we assign a linearly interpolated credibility score $c \in [0; 1]$. There is no punishment if the assessment took longer than the average.

D. Aggregation of Ratings

Quality is a viewpoint-dependent characteristic. As we will present in Section IV, we found significant differences between the expert assessments. To avoid biasing the results towards the opinion of one expert, we aim to find a consensus between them. Reaching a consensus usually involves convincing others with arguments. But we refrain from finding a consensus per discussion as proposed by Rosqvist [15]. To save time, we target a fully automatable process. In this paper, we use the term consensus to refer to the result of a weighted vote. Since the study participants have varying backgrounds and experiences, it appears natural to not treat every submission equally. Instead, we use an aggregation that assigns weights to each rating. The result of the aggregation is then considered the consensus. The Maximum Likelihood Expectation Maximization Algorithm (EM algorithm) is a statistical approach to determine such unknown weights iteratively [34]. Dawid and Skene [39] popularized this algorithm for problems where different observers may report different interpretations of the same yet unknown classification. They use the example of several clinicians diagnosing a patient, which transfers directly to our study where analysts diagnose a code snippet. In both cases the true label is unknown. The algorithm jointly maximizes the likelihood of experts' error rates, i.e., their reliability, and calculates the most probable classification of the code or patient, respectively. This way, the algorithm determines the consensus of the evaluators. The error rates are stored and later on reported to the users as part of a gamification approach. The quality of the aggregation result depends on the chosen starting values of the reliabilities [34], [39]. The initial weights incorporate knowledge from the validation steps described in Section III-C. The product of the plausibility score p and credibility score c forms the first estimate of the weight.

IV. RESULTS

In total, 70 experts participated in this study and submitted 1976 ratings. Eventually, the labeled dataset consists of 519

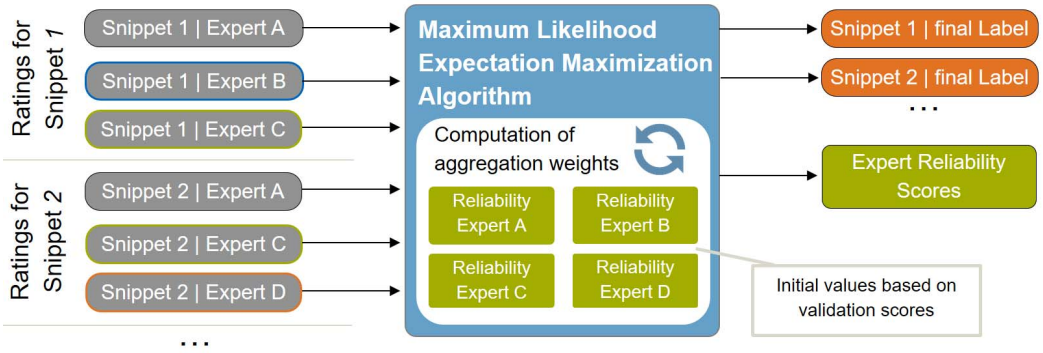

Fig. 5. The rating aggregation approach.

distinct code snippets. Here, we only consider code that received three valid ratings.

A. Influences on the Perceived Maintainability

Strong evidence was found that the understandability of code is valued more than its readability, complexity, or modularization. In our study, every submitted evaluation comprises an overall maintainability label, a label for each of the four subdimensions and an optional comment. Every label is a rating on a four-part Likert-scale.

The correlation between the perceived maintainability and its subaspects is tested using Pearson's Correlation Coefficient [40]. For this analysis, we convert every label to an integer value between 1 and 4. The highest coefficient was observed for understandability (0.80), followed by modularization (-0.74), complexity (-0.73) and readability (0.72). The questionnaire uses an inverted scale for modularization and complexity, thus their correlation is negative. Notably, all scores lie in a similar range. To refine the results, we also apply the Relief algorithm modified for regression [41], [42]. This approach confirms understandability (0.18) as the most influential subaspect. Readability (0.11), modularization (0.07) and complexity (0.04) seem to be less expressive predictors. If we treat the labels as nominal values instead of ordinal ones, the results are still valid. We analysed the influence of the subdimensions using information gain [43], information gain ratio [43], and the performance of a simple one rule classifier using only that single feature [44]. All three approaches confirm the previous result and rank understandability higher than the other three attributes.

The labeling platform provides the opportunity to submit free text comments about the assessed code. This option was used 198 times. A majority of participants used it to summarize the evaluated code snippet or their assigned labels. Other prominent topics were content or domain-specific remarks arguing about the specific implementation of a function. Inconsistencies within the code snippet and hardcoded values were perceived to hinder maintenance. Concerns regarding the violation of coding conventions were also widespread. Another recurring theme were comments in the evaluated code. Missing comments in places where the code is difficult to understand without further explanation was often perceived as negative. Code in comments, bad quality of comments, and *'todo'* or *'fixme'* annotations were found irritating, too. Furthermore, some participants felt distracted by long copyright statements.

B. Dissent and Consensus between Experts

We find that disagreement between experts happens often and significantly. In our study, every code snippet was assessed by at least three experts. There exist 2872 distinct rating pairs, i.e. pairs of ratings for the same code by different participants. In total, we find disagreement in 2107 of these pairs (73.4%). This expresses that the experts disagreed in at least one observed aspect. To distinguish between negligible and significant dissent, we calculate the sum of the differences. Please note that there were five questions asked. A cumulative deviation greater than five therefore means that the individual ratings deviate on average by more than one point in each question. We observe such significant differences in 493 cases (17.2%). In 36 cases (1.2%), the difference was even greater than ten. This threshold corresponds to an average deviation greater than two in each question. In one actual example, expert *A* assigned the ratings *[1, 1, 3, 4, 1]*, while expert *B* rated almost completely the opposite: *[4, 3, 1, 1, 4]*. This example will be examined further in the discussion. Similar dissent can be found in many instances. One snippet, e.g., was evaluated by four experts. Two experts agree in pairs, but their opinion contradicts that of the other pair. Not only does the evaluation of the single aspects vary, but we also identify differences regarding the final judgment. Experts assign identical ratings for the subdimensions but differ in the overall judgment. We noticed this in 24 cases.

C. The Software Maintainability Dataset

To overcome the observed disagreement, a consensus should be reached. In this study, the participants were not available for discussions, therefore, the consensus has to be found differently. We apply the EM algorithm to aggregate the opinions of several participants. An in-detail description can be found in Section III-D. The algorithm computes the probability for each rating to be correct. For the remainder of this section, we interpret the class with the highest probability as the final

TABLE III
DISTRIBUTION OF LABEL 'OVERALL MAINTAINABILITY' PER PROJECT

Project	strongly agree	weakly agree	weakly disagree	strongly disagree
ArgoUML	34	25	11	4
Art of Illusion	10	20	25	18
Diary Management	7	2	2	0
JSweet	63	5	2	3
JUnit 4	60	12	1	0
xApp	21	10	3	1
xBackend	18	11	5	1
xDispatch	32	24	10	7
xPrinting	31	21	14	6
Across Projects	**276**	**130**	**73**	**40**

label. Table III shows the distribution of the final labels for the overall maintainability of every study project. The last row of the table summarizes the distribution across all projects. From this data, it can be seen that positive assessments outweigh negative evaluations. In total, 78% of the examined files are reviewed as positive or very positive. Only 22% of the classes are considered hard or very hard to maintain. In 8 out of 9 projects the majority of the assessed classes are perceived as easily maintainable. Then, the number of classes in each category diminishes with decreasing maintainability. The Art of Illusion project is the only exception to this observation. Here, the majority of the files are considered difficult to maintain. Fig. 6 visualizes the overall distribution across all projects. These pie charts also show the distribution across all open source projects in comparison to the closed source projects. In total, the dataset includes 304 open source classes (59%) and 215 classes from closed source systems (41%). The evaluated open source code and its labels are available at [33].

V. DISCUSSION

A. Discussing the Results

RQ1: Our study showed that understandability has the highest impact on the overall maintainability of software. This finding is of immediate interest to software developers since understandability is an actionable characteristic. Developers can actively aim to improve this attribute of their code and thus improve its maintainability as such. We would like to refer once again to the definition of modularity and understandability as it is applied here. In this study, the term modularity is used to describe whether the class and its content is of adequate size. Understandability refers to the ease to identify concepts behind the code and identify where a certain concept is implemented. Our results do not justify completely neglecting any of the examined characteristics. All examined aspects show a correlation with maintainability and can thus be considered reasonable subcharacteristics. However, trade-offs should be in favour of understandability.

RQ2: Missing necessary comments or comments of bad quality were found to decrease the maintainability. This confirms that comments are important to comprehend code [45]. Also, it supports our finding that understandability is the most important quality characteristic. The negative statements about commented-out code support this, too. There are several reasons why code is stored in comments and not deleted [46]. However, the participants are afraid this code has to be analysed thoroughly to figure out why it has not been deleted. Since the code was taken from released products, *'todo'* and *'fixme'* annotations were perceived negatively for similar reasons. Here, the maintainer must analyse whether the issue was resolved or the code was released with incomplete features.

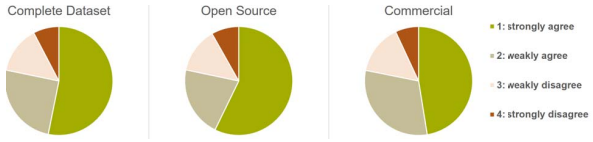

Fig. 6. The distribution of the label 'overall maintainability' per project type.

RQ3: Although the participants of this study are qualified software analysts, engineers, or researchers and are affiliated with renowned companies, their judgment is not unanimous. We found the dissent within a group does not deviate much from the dissent within all participants. However, we found exceptions when focusing on *significant* dissent. Within one company, we observe significant dissent in only 9.9% of the cases, while across groups it occurs in 17.2%. Other user groups do not show such deviations. One explanation could be that dissent is mostly caused by personal preferences. Another explanation might be bias introduced by various domains. A software engineer who is mostly concerned with automotive software might evaluate code from the insurance domain differently than an engineer who is familiar with the topic. To make reliable statements about the influence of the domains, we would need a larger amount of ratings per snippet. Otherwise, it is hard to reason that differences occurred due to the domain and not by personal preferences.

To overcome the observed dissent, a consensus has to be found. One instance, for example, was rated almost opposite by some experts. The snippet contains both complex and simple methods. Variable names are not self-explanatory, but the code is well documented. In fact, one can find arguments for positive and negative statements in every rating dimension. This instance exemplarily shows why it is important to investigate expert evaluations in depth.

RQ4: The EM algorithm dynamically aggregates the experts' ratings. Using predefined weights is not possible since we do not know up front which experts are more reliable than others. We refrained from using the job description and experience of a participant here for three reasons: First, this method can easily be manipulated by spammers. Second, it is not trivial to define how, for example, five years of experience as a developer should count against three years as an analyst. Third, the approach suffers from the assumption that people gain expertise through age only. While discussing the idea with industrial partners, we learned that long years of experience are not necessarily an indicator of higher reliability. Therefore, the initial reliability score does not incorporate this information.

B. Discussing the Dataset and Its Usage in Future Work

The created dataset can be used to design tools that predict software maintainability. Furthermore, our dataset contains data about the perceived understandability, readability, adequate size, and complexity as well. Instead of aiming at maintainability as such, future work can also focus on the automatic assessment of these attributes.

One of the most interesting characteristics of the created maintainability dataset is its unbalanced distribution. The majority of the files were evaluated to be easily maintainable, while only a few files were actually negatively perceived. This is not an ideal basis for building classifiers. However, subsequent trimming of the data to a balanced form would lead to too few data points to make reliable statements. Furthermore, that trimmed dataset would not be representative anymore. Nevertheless, the reported label distribution has implications on the design of approaches to automate assessments. In our sample, only 22% of the Java classes are considered problematic. In software quality assessments, it is important to precisely identify hotspots of bad quality. The challenge now is to fabricate tools that identify these rare but important instances reliably. The evaluation of tools should therefore give special thought to the classification of these cases instead of aiming for high overall precision and recall.

Although the label distribution is similar in 8 of the 9 analysed systems, it deviates considerably in the Art of Illusion project. This emphasizes the importance to consider data from several projects to base tool development on. Additional studies can add more labels from further projects and improve the generalizability of the results. Another progression of this work is to compare the expert evaluations to the observed maintenance effort. This work focuses on the maintainability of software. Further studies focusing on other dimensions can contribute to a holistic software quality dataset.

C. Discussing the Methodology: Threats to Validity

To fully exploit the potential of prediction algorithms, a well-researched dataset is needed. All studies reviewed so far, however, suffer from at least one of the following drawbacks: They consider only obsolete programming languages [7], use a formal definition as the ground truth [8], keep the data confidential and hinder replicability of the results [17], [19], or lack generalisability due to a small number of sample projects and experts [17], [19]. To mitigate the effect of biased evaluators, we collaborated with 70 participants from 17 different companies. However, only selected participants were allowed to inspect commercially developed software. To avoid domain or project-specific bias, we included code from 9 projects, both open and closed source. Still, the analysed code is only written in Java. Limiting the study to one language eliminates the need to frequently accustom to new contexts. We chose Java because of its high relevance in industry.

The prioritization impacts which classes are labeled first and are thus part of the dataset. This is necessary because of the size of the corpus. Given enough time, the prioritization would not affect the created dataset at all. The algorithm is based on code metrics. Their correlation with quality is known from previous work [47]. However, the problem to predict maintainability from metrics is not yet solved. To build useful tools, the dataset has to contain all typical constellations of metrics. The clustering groups data with similar constellations together. Iterating through these clusters ensures that representatives of all typical constellations are selected.

The labeling platform presents code snippets to experts who evaluate the code. The granularity of Java classes was chosen to keep the labeling effort feasible. This implies that only intra-class characteristics can be evaluated. We are aware that major aspects of software quality are inter-class attributes such as cloning and coupling. These are not characteristics of one isolated code snippet, but a class and its wider context. We limited the scope of the assessments to intra-class characteristics to foster the practicality and scalability of the labeling. Had we included the context of a class in the assessment, the labeling would have been far too time consuming and complex to collect a reasonable amount of labels. An interesting continuation of this work would be to include more context and inter-class relationships.

The labeling platform only displays the code and its location inside the package structure of the project. One could argue that static analysis metrics are used in quality assessments in practice [3] and should therefore be presented here as well. Actually, we desist from that to avoid biasing the results. The goal is to capture the experts' opinions without any other influences. To display metrics can take the focus away from the code itself. Experts might be biased by implicit thresholds, e.g. for the size of a class, and draw conclusions based on that metric without actually reading the code. In fact, showing metrics might lead to the wrong assumption that they *must* be taken into account. The selection which metrics to display and which not might consequently bias the labeling even more. Most static measurements such as, e.g., nesting depth can be observed directly from the code as well. One notable exception that is indeed hard to identify manually is intra-class cloning.

Most existing manually labeled quality datasets rely on one single expert and his evaluation. Therefore, that data is highly biased towards that expert's subjective opinion. In our study, every code snippet was evaluated by three participants. A snippet may receive more than three ratings if several experts are evaluating the same code snippet simultaneously.

Besides an overall judgment, we ask participants to assess four selected subaspects of quality. The selection of these dimensions is discussed in detail in Section II-C. The analysis of the submitted comments showed which aspects the participants took into account in the evaluation. However, the analysis could only respect those aspects that the experts found worth mentioning. Therefore, the list is not exhaustive.

VI. Related Work

Software quality is subject to many influences. For example, Naggapan et al. showed the correlations between organizational structures and software quality [48]. Previous research established that software quality consists of several subdimensions, such as maintainability [9], [10]. The ISO/IEC standard on software quality [9] deliberately does neither define weights to aggregate the subcharacteristics nor how to measure them. One method to reason about the quality of software is to rely on expert judgment. Code reviews can help to evaluate and control the quality of programs [4], [5]. Expert-based assessments are subjective to the expert conducting it. Rosqvist et al. [15] propose to incorporate the uncertainty of experts and the level of consensus between them.

Though the automatic extraction of metrics is arguably faster than manual reviews, there are also drawbacks of such tools. Several studies point out that automated tools are prone to false positives [49], [50]. Consequently, the metrics have to be put into context [3] and interpreted by a human expert [6]. Benestad et al. [6] even advocate clear-cut strategies for selection, aggregation, and interpretation of the collected metrics. One promising approach to combine the advantages of automation and reviews is supervised machine learning. Li and Henry [7] published an often used software maintainability dataset. It consists of object-oriented metrics they extracted from programs written in Classic-Ada. Here, the maintainability of a program class is defined by the number of changed lines in this class. Van Koten and Gray trained Bayesian Networks on this data [51]. Kumar et al. applied neuro-genetic algorithms [26], while others use regression-based models [7], [18]. Misra [8] performed a study using automatically labeled data. He used the Maintainability Index [52] to express the maintainability of code. However, the relation of these labels and the maintainability as perceived by humans remains speculative. Other researchers base their approaches on expert evaluations. Hayes and Zhao use the perceived maintainability [53] but apply their approach only to systems from student classes. In contrast, Pizzi et al. [17] used data from one real-world system. However, only one expert labeled the files. The same limitation is found in the study of Hegedűs [18]. Schnappinger et al. [19] had three experts evaluating code, but do not report the exact procedure.

However, it is important to define which aspects the evaluators considered in the assessment. The correlations between different quality characteristics was examined by Jung et al. [16]. However, the participants of their study were end-users who rated a software product. Correia et al. [54] examined the correlations between system properties and quality characteristics. They surveyed three experts and defined the median as the consensus. In contrast, Tokmak and colleagues [55] hint study participants should not be treated equally. Their study indicates that experts and novices tend to evaluate software differently. However, their work focuses on quality in use.

In summary, little research was done on how experts perceive quality and how they value different characteristics. This is problematic since quality datasets are crucial to develop effective prediction models. So far, often datasets are used that do not consider expert reviews though they seem to be superior to other approaches. In contrast, other studies follow the experts' assessment without further analysis. The study presented in this paper closes this gap. Our research forms a solid base on which future attempts to automate quality assessments can build. We collect manual labels from a large and diverse expert group, examine the evaluations thoroughly, and identify which aspects of maintainability the experts took into account.

VII. Conclusion

This study investigates expert evaluations of software maintainability and creates a robust maintainability dataset that can be used to develop reliable prediction tools. In our survey, 70 professionals assessed code from 9 open and closed source software projects. The submissions included ratings of the readability, understandability, complexity, modularity, and overall maintainability of the code. The projects at hand contain more than 15,000 files and account for 1.43 million source lines of code. A sophisticated prioritization algorithm defined the order in which the data points were labeled. This keeps the dataset both representative and insightful. The resulting dataset contains the consensus assessment of 519 Java classes. Interestingly, our work revealed that disagreement between experts occurs frequently and considerably. Although we narrowed down the number of perspectives the code has to be evaluated from, we found significant dissent in 17% of the cases. Small deviations are observed in 73%. Consequently, we argue that a consensus between the experts has to be found before relying on their evaluations. For this reason, we presented an aggregation algorithm. Based on the submitted ratings and the evaluators' error probabilities, it determines which judgment is most probably correct. In 8 out of 9 study projects, the majority of the files were considered to be easily maintainable. Our analysis of the assessments revealed that understandability has the highest impact on the overall perceived maintainability. Whilst this attribute has the highest influence neither readability nor complexity nor modularity should be neglected. Furthermore, we identified which other aspects of the code the experts took into account. Among the most reported issues are violations of coding conventions, commented-out code, *todo* or *fixme* annotations, and missing or unhelpful comments.

Until now, most other software quality datasets are either not based on expert judgment or rely on a small group of experts without further analysis. This paper, however, provides in-depth insights on how experts perceive quality. Moreover, we showed that this perception can vary and that aggregating ratings is therefore necessary and useful. Finally, we present a robust dataset as the basis for building precise and useful quality assessment tools. In the spirit of open science, the evaluated open source code and its ratings are shared with the scientific community.

REFERENCES

[1] 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.

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

[3] 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.

[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. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 202–211.

[6] 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.

[7] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, pp. 111–122, 1993.

[8] S. C. Misra, "Modeling design/coding factors that drive maintainability of software systems," *Software Quality Journal*, vol. 13, no. 3, pp. 297–320, 2005.

[9] ISO/IEC, "ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[10] B. W. Boehm, J. R. Brown, and H. Kaspar, "Characteristics of software quality," 1978.

[11] R. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[12] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 121–130.

[13] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 73–82.

[14] D. A. Garvin, "What does - product quality - really mean," *Sloan management review*, vol. 25, 1984.

[15] T. Rosqvist, M. Koskela, and H. Harju, "Software quality evaluation based on expert judgement," *Software Quality Journal*, vol. 11, no. 1, pp. 39–55, 2003.

[16] H.-W. Jung, S.-G. Kim, and C.-S. Chung, "Measuring software product quality: A survey of iso/iec 9126," *IEEE software*, vol. 21, no. 5, pp. 88–92, 2004.

[17] N. J. Pizzi, A. R. Summers, and W. Pedrycz, "Software quality prediction using median-adjusted class labels," in *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No. 02CH37290)*, vol. 3. IEEE, 2002, pp. 2405–2409.

[18] 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.

[19] M. Schnappinger, M. H. Osman, A. Pretschner, and A. Fietzke, "Learning a classifier for prediction of maintainability based on static analysis tools," in *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 2019, pp. 243–248.

[20] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE transactions on software engineering*, vol. 30, no. 4, pp. 246–256, 2004.

[21] N. Ahmad and P. A. Laplante, "Employing expert opinion and software metrics for reasoning about software," in *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*. IEEE, 2007, pp. 119–124.

[22] N. Khamis, J. Rilling, and R. Witte, "Assessing the quality factors found in in-line documentation written in natural language: The javadocminer," *Data & Knowledge Engineering*, vol. 87, pp. 19–40, 2013.

[23] W. Zhang, Y. Yang, and Q. Wang, "Network analysis of oss evolution: an empirical study on argouml project," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, 2011, pp. 71–80.

[24] P. Schugerl, J. Rilling, and P. Charland, "Mining bug repositories–a quality assessment," in *2008 International Conference on Computational Intelligence for Modelling Control & Automation*. IEEE, 2008, pp. 1105–1110.

[25] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Unsupervised learning for expert-based software quality estimation." in *HASE*. Citeseer, 2004, pp. 149–155.

[26] L. Kumar, D. K. Naik, and S. K. Rath, "Validating the effectiveness of object-oriented metrics for predicting maintainability," *Procedia Computer Science*, vol. 57, pp. 798–806, 2015.

[27] W. E. Saris and I. N. Gallhofer, *Design, evaluation, and analysis of questionnaires for survey research*. John Wiley & Sons, 2014.

[28] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[29] T. Gardner-Medwin and N. Curtin, "Certainty-based marking (cbm) for reflective learning and proper knowledge assessment," in *Proceedings of the REAP International Online Conference: Assessment Design for Learner Responsibility. Glasgow: University of Strathclyde*, 2007, pp. 29–31.

[30] D. R. Raymond, "Reading source code." in *CASCON*, vol. 91, 1991, pp. 3–16.

[31] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[32] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software complexity and maintenance costs," *Communications of the ACM*, vol. 36, no. 11, pp. 81–95, 1993.

[33] M. Schnappinger, A. Fietzke, and A. Pretschner, "A software maintainability dataset," Sep 2020. [Online]. Available: https://figshare.com/articles/dataset/_/12801215

[34] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[35] U.-D. Reips, "Standards for internet-based experimenting." *Experimental psychology*, vol. 49, no. 4, p. 243, 2002.

[36] N. Malhotra, "Completion time and response order effects in web surveys," *Public Opinion Quarterly*, vol. 72, no. 5, pp. 914–934, 2008.

[37] F. Keller, S. Gunasekharan, N. Mayo, and M. Corley, "Timing accuracy of web experiments: A case study using the webexp software package," *Behavior research methods*, vol. 41, no. 1, pp. 1–12, 2009.

[38] J. M. Ihme, F. Lemke, K. Lieder, F. Martin, J. C. Müller, and S. Schmidt, "Comparison of ability tests administered online and in the laboratory," *Behavior Research Methods*, vol. 41, no. 4, pp. 1183–1189, 2009.

[39] A. P. Dawid and A. M. Skene, "Maximum likelihood estimation of observer error-rates using the em algorithm," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 28, no. 1, pp. 20–28, 1979.

[40] K. Pearson, "Vii. note on regression and inheritance in the case of two parents," *proceedings of the royal society of London*, vol. 58, no. 347-352, pp. 240–242, 1895.

[41] K. Kira and L. A. Rendell, "A practical approach to feature selection," in *Ninth International Workshop on Machine Learning*, D. H. Sleeman and P. Edwards, Eds. Morgan Kaufmann, 1992, pp. 249–256.

[42] M. Robnik-Sikonja and I. Kononenko, "An adaptation of relief for attribute estimation in regression," in *Fourteenth International Conference on Machine Learning*, D. H. Fisher, Ed. Morgan Kaufmann, 1997, pp. 296–304.

[43] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to data mining*. Pearson Education India, 2016.

[44] R. Holte, "Very simple classification rules perform well on most commonly used datasets," *Machine Learning*, vol. 11, pp. 63–91, 1993.

[45] J. Raskin, "Comments are more important than code," *Queue*, vol. 3, no. 2, pp. 64–65, 2005.

[46] Y. S. Yoon and B. A. Myers, "An exploratory study of backtracking strategies used by developers," in *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2012, pp. 138–144.

[47] 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.
[48] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 521–530.
[49] 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.
[50] 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.
[51] C. Van Koten and A. Gray, "An application of bayesian network for predicting object-oriented software maintainability," *Information and Software Technology*, vol. 48, no. 1, pp. 59–67, 2006.
[52] 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.
[53] J. H. Hayes and L. Zhao, "Maintainability prediction: A regression analysis of measures of evolving systems," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 601–604.
[54] J. P. Correia, Y. Kanellopoulos, and J. Visser, "A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 61–70.
[55] H. S. Tokmak, L. Incikabi, and T. Y. Yelken, "Differences in the educational software evaluation process for experts and novice students," *Australasian Journal of Educational Technology*, vol. 28, no. 8, 2012.