Empirical Software Engineering (2023) 28:128
https://doi.org/10.1007/s10664-023-10381-0

Developers talking about code quality

Jürgen Börstler[1] · Kwabena E. Bennin[2] · Sara Hooshangi[3] · Johan Jeuring[4] · Hieke Keuning[4] · Carsten Kleiner[5] · Bonnie MacKellar[6] · Rodrigo Duran[7] · Harald Störrle[8] · Daniel Toll[9] · Jelle van Assema[10]

Accepted: 9 August 2023 / Published online: 21 September 2023
© The Author(s) 2023

Abstract
There are many aspects of code quality, some of which are difficult to capture or to measure. Despite the importance of software quality, there is a lack of commonly accepted measures or indicators for code quality that can be linked to quality attributes. We investigate software developers' perceptions of source code quality and the practices they recommend to achieve these qualities. We analyze data from semi-structured interviews with 34 professional software developers, programming teachers and students from Europe and the U.S. For the interviews, participants were asked to bring code examples to exemplify what they consider good and bad code, respectively. Readability and structure were used most commonly as defining properties for quality code. Together with documentation, they were also suggested as the most common target properties for quality improvement. When discussing actual code, developers focused on structure, comprehensibility and readability as quality properties. When analyzing relationships between properties, the most commonly talked about target property was comprehensibility. Documentation, structure and readability were named most frequently as source properties to achieve good comprehensibility. Some of the most important source code properties contributing to code quality as perceived by developers lack clear definitions and are difficult to capture. More research is therefore necessary to measure the structure, comprehensibility and readability of code in ways that matter for developers and to relate these measures of code structure, comprehensibility and readability to common software quality attributes.

Keywords Code quality · Semi-structured interviews · Software development · Source code properties · Quality perception

Communicated by: Venera Arnaoudova

✉ Jürgen Börstler
jub@bth.se

Extended author information available on the last page of the article

1 Introduction

Code quality is a central factor in software development and there is an abundance of advice on how to write “good code,” e.g. (Green and Ledgard 2011; Martin 2008; Sutter and Alexandrescu 2004; Vermeulen et al. 2000). Despite a large body of work on software quality measurement (Fenton and Bieman 2014; Nuñez-Varela et al. 2017), there is little conclusive evidence on the relationship between software quality measurements and common software quality attributes (Jabangwe et al. 2015; Jørgensen 1999). Furthermore, many common software measures loose their predictive power when controlled for program size (Gil and Lalouche 2017; Tahir et al. 2018) and do not capture quality improvements as perceived by software developers (Al Dallal and Abdin 2018; Pantiuchina et al. 2018).

Research also shows that quality issues at the code level increase the cognitive load of developers (Fakhoury et al. 2018; Müller and Fritz 2016), which affects their performance negatively (Ko and Myers 2005; Sweller et al. 2011). Since a considerable amount of effort is dedicated to software quality during its development (Menzies et al. 2006), a better understanding of developers’ perceptions of software quality on code level might help to utilize this effort more effectively.

In this study, we investigate how software developers perceive code quality, which quality aspects they perceive as more or less important and what they recommend to achieve high code quality. The study uses interview data from 34 professional software developers, programming teachers and students with an average of 5.8 years of experience as professional software developer, which we categorized into three groups based on their number of years of experience as professional programmers. The interviews were carried out in the context of an earlier study (Börstler et al. 2018), but the majority of the data used here has neither been coded nor analyzed before (see Section 3.2 for details).

This study makes the following contributions.

1. A qualitative study providing insights on aspects of code quality that are considered important by developers.
2. A study design package to facilitate the design and execution of similar studies.
3. An anonymized data-set for further analysis.

The organisation of the paper is as follows. Section 2 discusses related work on defining software quality. The background and experimental methodology are presented in Section 3. Empirical results are presented and analyzed in Section 4. A discussion of the findings follows in Section 5. Section 6 summarizes and concludes the paper. Information about the study design package and the anonymized data-set can be found in Appendix B.

2 Related Work

Over the years, several software quality models and standards have been developed to provide frameworks for defining and evaluating software quality (Nistala et al. 2019; Sadeghzadeh Hemayati and Rashidi 2018). The most recent ones are ISO/IEC 25010 (ISO/IEC 2011) and ISO 5055:2021 (Curtis et al. 2022). Whereas ISO/IEC 25010 defines quality in terms of eight quality characteristics, e.g., maintainability, ISO/IEC 5055 defines quality in terms of “weaknesses” that threaten the software’s reliability, security, performance efficiency, and maintainability. Although ISO/IEC 25010 (and its predecessors) provide definitions for the quality characteristics (and their sub-characteristics), Bakota et al. (2011) argue that the quality models do not deal with ambiguity which makes it hard to measure these characteristics.

A systematic mapping study on source code metrics (Nuñez-Varela et al. 2017) covering 2010–2015 identified 226 studies using more than 300 source code metrics in total. Although many of those metrics have been around for a long time, there still is little empirical evidence that connects source code metrics to software quality attributes. Pantiuchina et al. (2018), for example, mined commits with messages that clearly showed that the developers aimed to improve the cohesion, coupling, complexity and readability of the underlying source code. Their analysis shows that most of the considered code quality metrics were not able to capture the quality improvements as perceived by the developers.

There is evidence, though, showing that there is a relationship between low level code features and code quality. Butler et al. (2010), for example, showed that flawed identifier names are associated with low code quality. Recent research in program comprehension, moreover, shows that misleading names are more problematic than meaningless names (Avidan and Feitelson 2017), but that specific one-letter variables still can convey meaning (Beniamini et al. 2017). It has also been shown that structural differences at code-level affect program understanding, for example, that for-loops are significantly harder to understand than if-statements (Ajami et al. 2017) and that "maintaining undisciplined annotations is more time consuming and error prone" than maintaining disciplined ones (Malaquias et al. 2017).

Several studies have focused on how refactoring improves code quality (Alshayeb 2009; Chatzigeorgiou and Manakos 2010; Peters and Zaidman 2012; Kim et al. 2014; Tufano et al. 2017; Al Dallal and Abdin 2018). Alshayeb (2009) quantitatively assessed the effect of refactoring activities on the external quality attributes adaptability, maintainability, understandability, reusability and testability and concludes that refactoring does not necessarily improve these quality attributes. Chatzigeorgiou and Manakos (2010); Peters and Zaidman (2012); Kim et al. (2014) examine historical software data to evaluate the lifespan and evolution of code smells and the usage of refactoring to remove these code smells. A large-scale study comprising a survey, interviews and quantitative analysis of historical versions of Windows 7 (Kim et al. 2014) revealed that only 5% of the studied modules were improved (low inter-dependencies and complexities) when they were refactored. Similarly, Tufano et al. (2017) studied the change history of over 200 open source projects and observed that only 9% of the code smells were removed as a result of refactoring operations. Al Dallal and Abdin (2018) corroborate the findings of Kim et al. and Tufano et al. that refactoring does not always improve code quality. This indicates that refactoring alone is not effective for "fixing" bad code.

Research on inspections shows a positive impact on quality (Bavota and Russo 2015; Kollanus and Koskinen 2009). Several studies have focused on code reviews to check and improve the quality of code submitted into a master code repository (Hatton 2008; Kononenko et al. 2016; Rong et al. 2014). Kononenko et al. (2016) investigated factors that influence the review time, review decision and review quality as perceived by Mozilla core developers. Their open coding of the survey questions identified "code quality" and "understanding code change/base" as the overall most important themes. An exploratory study by Sadowski et al. (2018) on modern code reviews at Google reveals that "the foremost reason for introducing code review at Google was to improve code understandability and maintainability." Google's mandatory code reviews involve manual readability approval of developers having readability certification (Wintus et al. 2020). This entails an expensive infrastructure. Reliable metrics for measuring or predicting readability would therefore have a significant positive impact on code quality and productivity. A recent review on modern code reviews (Davila and Nunes 2021) corroborates that "code comprehension has been the main challenge faced by developers when reviewing a code change."

The literature lacks studies exploring practitioners' opinions and there are few studies on lower-level aspects of source code quality, like the one presented here. Bakota et al. (2011) suggest a probabilistic approach for computing maintainability (based on the definition in ISO/IEC 9126, the predecessor of ISO/IEC 25010) with the help of input from 28 experts. An evaluation on two software systems showed that their results aligned with the expectations of experts. Correia et al. (2009) interviewed 22 experts consisting of software engineers, software quality consultants and researchers to define a mapping from 9 system properties (e.g., system size and unit complexity) to the maintainability sub-characteristics of ISO/IEC 9126 and observed that there was no consensus on the mapping. Gorla and Lin (2010) explored organizational, individual and technological factors that impact IS project managers' perception of five software quality attributes (reliability, ease-of-use, maintainability, usefulness, and relevance). Based on the questionnaire results from 112 IS project managers, the authors conclude "that organizational factors are more important than technological factors in impacting software quality in IS projects." Antinyan et al. (2017) carried out an online survey with 100 respondents from industry and academia to investigate how much certain code characteristics affect 4 predefined internal quality attributes as well as maintainability time. They conclude that lack of structure and nesting depth substantially increase code complexity, which subsequently influences the maintenance time of code.

Taken together, existing research may benefit from a better alignment between software developers' perceptions of quality and the measures used to assess software quality. In this work, we therefore want to investigate software developers' discourse about code quality and their perceptions of code quality. Our study is unique in asking for practitioners' opinions unconditionally, i.e., without a preconceived definition or opinion about code quality.

All studies discussed in the paragraph above are mainly confirmatory and most are focused on maintainability. Except Bakota et al., they all suggest predefined models or definitions of quality characteristics and use close questions. In Bakota et al., the experts are involved in defining weights for the model. Of the four studies, two are interview-based studies with fewer respondents than our study. Furthermore, none of the four studies uses open questions and open coding.

3 Method

Our overall goal was to investigate the perceptions of code quality held by software developers and in which ways these perceptions vary with respect to the developers' background and experience.

3.1 Research Questions

In this research, we follow an exploratory approach to elicit software developers' perceptions of code quality unconditionally, i.e., without preconceived definitions or hypotheses about quality. We are interested in developers' "abstract" or general views about code quality (when they talk about code quality in general terms) as well as their "concrete" views when they reason about the properties of specific code examples.

In our earlier work (Börstler et al. 2018), we focused on participants' "abstract" perceptions of code quality. Here, we investigate in more detail how participants talk and reason about

code quality when referring to actual code examples, and in which ways this "concrete discourse" differs from their "abstract perceptions".

RQ1: Which abstract perceptions of code quality do software developers have?
RQ1.1: Which properties do they focus on when defining code quality?
RQ1.2: Which properties and practices do they focus on when suggesting quality improvements?
RQ2: Which properties related to code quality do software developers focus on in actual example code?
RQ3: In which ways do developers' "abstract" perceptions of code quality (see RQ1) differ from the specific properties they find relevant in actual code examples (see RQ2)?

3.2 Interview Questions

When designing the interview questions, we took an exploratory approach to get a better understanding of the perceptions of code quality. Our goal was to explore perceptions of code quality, not to test preconceived hypotheses. The interview questions were therefore framed very carefully, so that they did not introduce bias or suggest certain answers.

The interview contained three parts comprising 11 questions in total, most of them with sub-questions: Part 1 (Q1–Q3) focused on demographics and the participants' background and experience; part 2 (Q4) focused on quality aspects of the example(s) of code that the participants were asked to bring to the interview; and part 3 (Q5–Q11) covered further questions on code quality that were independent from Q4, and the example(s) discussed there. All questions in part 2 and part 3 (except Q6) were completely open.

Except for Q4, the questions were short free-text, numeric, or Likert-type questions. All Likert-type questions used a 7-item scale where only the end values were named explicitly. For Q4, we asked the participants to bring along example code or code snippets from their personal experience that they consider exhibit high or low quality. The rationale for this was to elicit how abstract properties or qualities of code manifest in actual first-hand code examples.

The full list of questions can be found in Appendix A. An overview of the study process and how this study relates to our earlier work (Börstler et al. 2018) can be found in Fig. 1. It should be noted that Q4 (the main part of the interview) as well as Q7 was neither coded nor analyzed in our previous work.

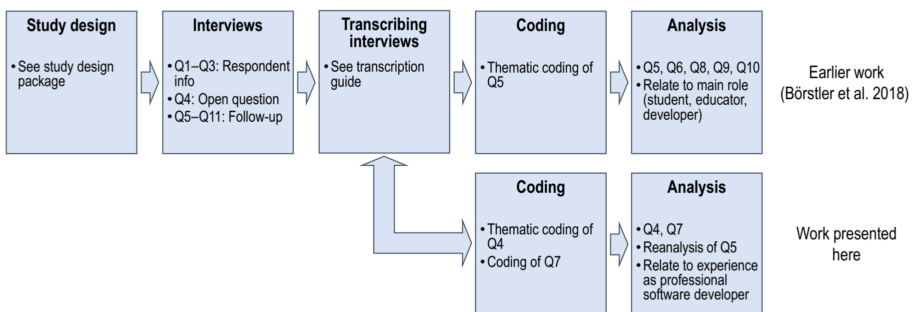


Fig. 1 Overview of the study process. The lower part of the figure shows the contributions of the work presented here in relation to our earlier work

3.3 Participant Recruiting/Sampling

Ten of the eleven authors participated in interviewing. The interviewers invited candidates to participate in the study by circulating an information sheet (see the "study design package" in Appendix B.1) in the channels available to him or her.

We invited students, educators, and professional software developers for participation in the study. Students should have completed several programming courses. Educators should have several years of teaching experience with courses covering programming, software design, or software quality. Professional software developers should deal with software development for a living, i.e. regularly read, write, test or review source code or low-level designs as a significant part of their work.

Initially, we collected basic information about potential participants (current occupation, education, level of professional programming experience, gender, and country) in a shared spreadsheet. The information in the spreadsheet was used by interviewees to facilitate recruiting of participants with varying backgrounds and experience.

According to our study guide (see Appendix B.1), we defined developers as "people who actually deal with software development for a living, i.e., people who regularly read, write, test or review source code or low-level designs." We targeted 3–5 interviews per interviewer to ensure a good geographical spread and minimize the risk for a common selection bias. The interviewers came from Sweden, Germany, USA, The Netherlands, and Finland. Other than the coordination described above, the individual interviewers applied convenience sampling.

3.4 Data Collection

We used a detailed interview guide with predefined and scripted questions. The interviews took 45–60 minutes and were either conducted in person or through video calls (using Google Hangouts, Skype, or Zoom).

The first 3 questions (Q1–Q3) were filled in by the interviewers. Question Q4, the main part of the interview, was recorded and transcribed. The remaining 7 questions (Q5–Q11) were filled in directly by the interviewees.

If the interviewees brought code examples in electronic form, we captured the screen to be able to connect the discussion to particular areas of the code. Otherwise, we made annotations on the print-outs to connect areas of code to statements in the interview.

For the transcription, we developed transcription guidelines to ensure a uniform transcription of the interviews. The guidelines were based on existing guidelines (Humble 2015) that we adapted to ensure a fluent transcript style. If possible, the interviews were held in the language the interviewees felt most comfortable in; either in a language where the interviewer was a native speaker or a language where the interviewer was highly fluent. All interview transcripts were translated to English before the analysis.

One of the co-authors of the interview script tested the questions using a pilot interview with a university teacher with professional programming experience. The pilot interview took 56 minutes and was recorded (both audio and computer screen). A part of Q4 was transcribed to test the transcription guide. After a discussion of the pilot, the co-authors of the interview guide revised the interview and transcription guides slightly. The data of the pilot study was discarded and not used for the analysis presented here. The interview script as well as the transcription guide are available for download and part of the study design package (see Appendix B.1).

3.5 Coding of Open Questions

We used open coding (Creswell and Creswell 2017) to categorize the data from the open questions.

In our earlier work (Börstler et al. 2018), two groups of co-authors independently extracted labels inductively from Q5 (definitions of code quality) and Q8 (factors/indicators of high quality), respectively. As a next step the initial labels from both groups were merged and generic categories were defined to encompass the specific data. When new categories emerged or their definition or labels changed, the whole dataset was re-categorized using the revised categories. Each label was connected to one category. This was done until all data had been categorized and no new categories emerged and/or were changed.

For the present work, we used these labels and categories as a starting point for the coding of Q4 and Q7 described below. For Q7 (recommendations for increasing code quality), we also categorized the answers into analytical and constructive approaches for quality improvement. Analytical approaches are based on assessing/observing the level of quality (e.g., static analysis), whereas constructive approaches deal with ensuring quality "by construction" (e.g., design patterns). Approaches that could not be categorized as either analytical or constructive were categorized as *Other*.

Question Q7 (recommendations for increasing code quality) was independently coded by four co-authors. Since Q7 added an activity aspect, new labels emerged and the original coding scheme was extended to accommodate those with as little as possible changes to existing labels and categories (see Section 4.3). The independent codings of Q7 were then merged at a common meeting together with the extended coding scheme. Differences and inconsistencies were discussed and resolved at the meeting. Inter-rater agreement (Krippendorff's alpha, computed via ReCal Freelon 2013) for the sub-questions of Q7 were between 0.633 and 0.815 which is considered a substantial agreement.

The extended coding scheme was then used for coding Q4. The coding was done in several steps to facilitate further analysis as described below.

1. Mark text passages relating code properties or coding activities to quality attributes.
2. Summarize each such passage by "assertions" of the following form:

$$< quality\ indicator \mid activity > \quad => \quad <effect>; <artifact>.$$

Detailed coding guidelines were described and then piloted independently by two of the co-authors. Two examples of such "assertions" and their corresponding text passages are shown in Fig. 2.

All interviews were then coded independently by two co-authors each, and the coding step was discussed at a common meeting. After the meeting the original coders discussed their independent codings and merged them into an agreed-upon version that was used for the analysis described in Section 4.6.

After this validation step, all assertions were imported into a spreadsheet for further analysis. Composite assertions, i.e., rows listing several quality indicators/ activities and/or effects, were broken down into multiple rows such that each combination of quality indicator/ activity and effect was placed on a separate row, i.e., a1, a2 => e1, e2 was replaced by a1 => e1, a1 => e2, a2 => e1, a2 => e2. Unclear or missing effects (e.g., "?" or "–") were replaced by the generic effect "higher (lower) quality," since this was the agreed-upon default for effects that were not made sufficiently specific by an interviewee. Ambiguities and questions regarding assertions were resolved with the help of the original interviewer.

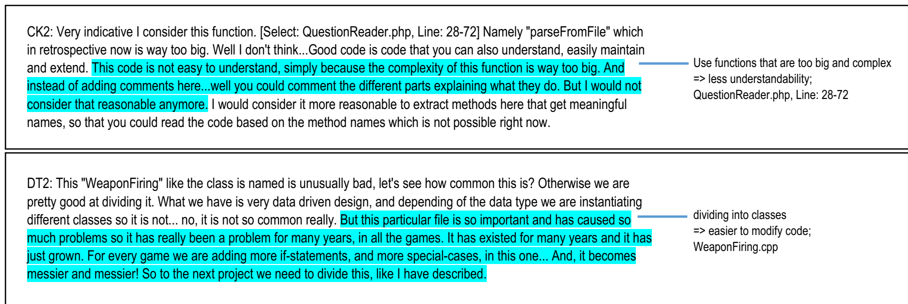


Fig. 2 Excerpts from the interview transcripts of interviewee CK2 (top) and DT2 (bottom) with marked passages and corresponding assertions to the right

The quality indicators/ activities (sources) and effects (targets) of all assertions were then categorized according to Table 3 by the first author and validated as follows:

1. Four interviews (of 34) were selected at random and categorized independently by four co-authors such that each interview's quality indicator/activity and effect were independently categorized by three co-authors each (including the first author). These four interviews corresponded to 86 of the 721 assertions (12%). Inter-rater agreement was moderate for the source categories (Krippendorff's alpha 0.434) and almost perfect for the target categories (Krippendorff's alpha 0.848).
2. The independent categorizations were then compiled into one document.
3. Inconsistencies were discussed and resolved by "majority vote" if necessary.
4. The first author revised the coding of the remaining interviews accordingly.

When coding Q4, the wording of an assertion was kept close to the wording of the interviewee, as shown in the examples in Fig. 2. As described above, the assertions were then categorized according to Table 3 to facilitate further analysis. This categorization could lead to assertions where source and target belong to the same category. An interview statement like "concise code leads to better readability" would, for example, be coded as concise code => readability and then categorized as readability=> readability since both, conciseness and readability belong to category *Readability*.

3.6 Threats to Validity

Internal validity is concerned with the study design, in particular whether the constructs used to answer the research questions are suitable and sufficient to answer the questions.

Two of the co-authors developed a thorough study guide including information about the sampling and recruitment of participants, a detailed interview script with lists of probing questions, as well as guidelines for transcribing the recorded interviews. This guide was discussed with the remaining authors to make sure that the questions are interpreted uniformly. Templates for the consent form and an information sheet for recruiting interviewees were also prepared.

The interview script contained closed and open questions. Question Q4 was intentionally left open and the interview guide encouraged interviewers to let the interviewees talk freely. Since interviewees brought code from their personal experience to the interview, we made sure that they could relate their personal perceptions about code quality to actual code examples. Respondents' discussions might have been different had they brought and talked about other

examples in other programming languages. However, we explicitly asked them "to bring along example code from your own personal experience" and to share "personal opinions regarding source code quality and to point out pieces of code that you consider have high or low quality." Since the study is exploratory with open questions, we do not consider this a threat to validity.

Interviewees without professional programming experience (students) were mostly self-selected, whereas study participants with professional programming experience were directly approached by the researchers. We targeted 3–5 interviews per interviewer and coordinated the recruitment of interviewees via a shared spreadsheet. Besides this coordination, the individual interviewers applied convenience sampling. Since the interviewers come from five different countries and each interviewer carried out only 3–5 interviews, we consider the risk for a common selection bias low.

During the coding of the transcripts, chances are that data is misinterpreted or coded wrongly. To mitigate this threat, we developed and piloted detailed coding guidelines (as described in Section 3.5). Question Q4 was coded in a two-step process. Step one was carried out independently by two co-authors and then validated. Step two was carried out by one author and then cross-validated independently by three co-authors with high inter-rated agreement. Question Q7 was coded independently of four co-authors with high inter-rater agreement. We therefore consider the coding reliable. Details about the coding and validation processes can be found in Section 3.5.

External validity is concerned with the generalizability of the results.

Since the sample size in this study is small and interviewees were limited to Europe and the U.S., we cannot generalize the results to software developers globally. Also, since the authors are primarily from academia, even most interviewees from professional software development have some relationship to academia. So, developers without any academic background might be underrepresented. However, within these limits we achieved a good spread of interviewees in terms of relevant experience. Our study design aimed for "theoretical saturation," i.e., a sample size where adding more interviews is unlikely to uncover further concepts or aspects. In a review on sample sizes for saturation in qualitative research (Hennink and Kaiser 2022) conclude that "saturation can be achieved in a narrow range of interviews (9–17)." With 34 interviews in our study, we believe that the risk for not having reached saturation is low. We have not validated, though, whether we reached saturation.

There is also a risk that the examples the interviewees brought along and discussed during the interviews are not representative of what developers actually care about in practice. To mitigate that risk, we used a standardized "interviewee information sheet" to ensure that all interviewees receive the same information (see the study design package in Appendix B.1). As discussed in Section 4.2 (Example Demographics), the interviewees brought many and varied examples. We have no indications of interviewees discussing topics that did not matter to them. We therefore consider this risk low.

4 Results and Analysis

4.1 Interviewee Demographics

In total, we analyzed data from 34 semi-structured interviews with students, educators, and professional software developers from 6 countries. Of the 34 interviewees, 22 (65%) have experience as a professional software developer (9.6 years on average, median 7.3 years) and

11 (32%) have software quality assurance as one of their responsibilities. The developers with software quality assurance as one of their responsibilities have 13.8 years (median 11 years) of professional experience on average. All students had completed at least 5 courses with significant programming components (8.9 on average). A summary of the basic demographics of the participants can be found in Table 1.

For our analysis, we categorized the interviewees along their experience as professional software developers (see column Cat-Exp in Table 1): Interviewees without experience as professional software developers (N), interviewees with some experience (S) but less than the median years of experience (7.3 years) and interviewees with extensive professional experience (E), i.e., more than the median years of experience.

4.2 Example Demographics

In total, 31 of the 34 interviewees brought along 130 code examples ranging from short code snippets on paper over links to guideline documents (containing example code snippets) to actual running code (single files to whole non-trivial projects). The remaining three did not bring code examples but referred to specific guidelines from a popular trade-book on code quality (Martin 2008). Fifteen interviewees discussed (among others) code from professional code bases, i.e., code that was written by professional software developers or code available from open source code bases. Twenty interviewees discussed bad as well as good examples. The examples were written in 14 different programming languages and 12 interviewees discussed examples from at least two languages, see Table 2.

4.3 Coding Scheme

In our previous work, we analyzed questions Q5 and Q8 by using codes based on key terms used in the answers and grouping them into related categories. These labels and categories were used as a starting point for the thematic coding of questions Q4 and Q7. While Q5 and Q8 focus on properties of code, Q7 asks for recommendations to improve code quality. The answers to Q7 therefore also refer to activities and not just specific code properties. The activities were eventually categorized as *Analysis* and *Generic*, respectively, as shown in the lower part of Table 3. The discussions around the open question Q4 also provided more varied answers than the answers to the more specific questions Q5 and Q8. This led to many extensions but also a few changes to our original coding scheme. We therefore also re-analyzed Q5 (see Section 4.4).

Some labels in category *Miscellaneous* are not properties of the code itself. However, since they have been mentioned in this context by the interviewees, we decided to keep them for reference. Similarly, labels in the *Non-specific* category are too general to gain additional insight. As before, we still kept them to provide a comprehensive picture of the responses. The total numbers in both categories are so small, that they do not affect our results.

4.4 Definitions of Code Quality (RQ1.1)

In question Q5, we asked interviewees (in writing) about their definition of software quality.

> *Q5*: How would you define code quality? Which properties, features or indicators show you, personally, something about quality?

Table 1 Basic demographics of the participants in the study

ID	Gender	Country	Role(s)[1]	Y-Exp[2]	Cat-Exp[3]	QA[4]
BM2	Female	USA	E	0	N	N
BM3	Male	USA	S	0	N	N
BM4	Male	USA	D	38	E	Y
CK1	Male	Germany	D,E	8	E	Y
CK2	Male	Germany	D	4	S	N
CK3	Male	Germany	S	0	N	N
DT1	Male	Sweden	E	0	N	N
DT2	Male	Sweden	D	17	E	Y
DT3	Female	Sweden	E	0	N	N
DT5	Female	Sweden	D	4	S	N
HK1	Male	The Netherlands	D,E	20	E	N
HK2	Male	The Netherlands	D	8	E	Y
HK3	Male	The Netherlands	D,S	0.5	S	N
HS1	Male	Germany	D	21	E	Y
HS2	Male	Germany	D,E	6.5	S	Y
HS3	Male	Germany	D	11	E	Y
HS4	Male	Germany	D	6	S	Y
JA2	Male	The Netherlands	D,S	1.25	S	N
JA3	Male	The Netherlands	D,S	1	S	N
JA4	Male	The Netherlands	D,E	1.5	S	N
JB1	Male	United Kingdom	D,E	14	E	Y
JB4	Male	Sweden	S	0	N	N
JB5	Female	Sweden	S	0	N	N
JJ1	Male	The Netherlands	D	2.5	S	Y
JJ2	Male	The Netherlands	E	0	N	N
JJ3	Male	The Netherlands	D	20	E	Y
RD1	Male	Finland	D,S	3	S	N
RD2	Male	Finland	S	0	N	N
RD3	Female	Finland	S	0	N	N
RD4	Male	Finland	D,E	2	S	N
RD5	Male	Finland	E	0	N	N
SH1	Male	USA	S	0	N	N
SH2	Male	USA	D,S	9	E	N
SH3	Female	USA	D,E	15	E	N

[1] **D**eveloper, **E**ducator, **S**tudent. Multiple roles are possible
[2]Experience as professional software developer in years
[3] Categorization of professional experience: **N**o experience, **S**ome experience, **E**xtensive experience
[4] Responsibility for quality assurance (**Y**es/**N**o)

The answers ranged from short lists of characteristics to more elaborate answers. The shortest answer was 8 words and the longest 79 words with a median of 22 words. We categorized the answers into 116 references to defining properties in total. On average, the answers referred to 3.4 defining properties (1–6; median: 3).

Table 2 Overview over the programming languages of the example programs discussed by interviewees (some interviewees discussed examples in several languages)

Interviewees	Language
14	Java
6	C#
5	Python
4	C++
4	JavaScript
4	PHP
3	Scala
2	C
2	SQL
1	ActionScript, Attribute grammars, Haskell, Julia, Ruby

Readability and *structure* were the most commonly used defining properties for code quality for our sample developers in total (see Fig. 3). Of the 34 developers 28 (82%) and 22 (65%), respectively, referred to *readability* and/or *structure*. All other properties were referred to by 27%–38% of our sample developers. When looking at subgroups of developers based on gender, level of experience as professional developer, or experience with responsibility for quality assurance, the rankings of properties are very similar (see Fig. 4). A notable difference is that experienced developers and those with QA responsibility rank *correctness* higher than the other groups. *Readability* and *structure* are consistently ranked as the topmost and second most important properties, respectively, by all subgroups.

The spearman rank correlations between groups are insignificant, except between the groups without (N) and some (S) professional experience in software development, where we found a high positive relationship (r_s=0.74, p=0.022).

Answer to RQ1.1: *Readability* and *structure* are the most common defining properties for quality code.

4.5 Properties Targeted by Quality Improvements (RQ1.2)

In question Q7, interviewees were asked to provide suggestions for improving code quality.

Q7: Please provide your top-3 recommendations for increasing the quality of code. Please indicate when a recommendation applies in special cases only.

Besides their top-3 recommendations interviewees could also provide “further important recommendations”. Interviewees provided 116 recommendations in total (2–4; average: 3.4).

When looking at the code properties respondents focused on within their recommendations for increasing code quality (Q7), we can see that *structure* and *readability* are again most commonly referred to (see Fig. 5). Due to the addition of categories *generic* and *analysis*, the answers for Q7 were distributed over more categories than the answers for Q5 and all “old” categories. Only category *documentation* increased its percentage. The group without experience from professional software development (N) accounted for most of the increase in category *documentation*.

Table 3 Categories and labels used for coding

Category	Labels
— quality indicators (properties of code) —	
Comprehensibility	comprehensible, understandable, clear purpose, clarity, simplify code, unusual syntax, unusual use of language features
Correctness	runnable/ free of bugs, language/ model/ framework choice, functionally correct (meeting business requirements), exception handling
Documentation	documented, commented, self-descriptive codes
Dynamic behavior	robust, good performance, secure, good use of time and resources (during code execution), optimize code
Maintainability	maintainable, adaptable, reusable, used by others, interoperable, portable, not hardcoding variables and input, not making errors later, global variables, using standard libraries/ functions, use config files, flexibility
Miscellaneous	license, suitable data structure, support, tools and libraries, delivery (in terms of rushing for it), trust, programming paradigm
Readability	readable, no useless code, brevity/ conciseness (small scale; Structure otherwise), formatting/ layout, style, indentation, adherence to naming conventions, code standard, mixing natural languages, usage of whitespace, use of idioms, consistency, no useless comments
Structure	well structured, modular, cohesion, low coupling, no duplication, decomposition, structural complexity, encapsulation, grouping, elegance, doing one thing
Testability	testable, test coverage, automated tests, data validation, I/O validation, debugging
Non-specific	higher/ lower/ good quality, no effect specified
— activities to improve quality —	
Analysis	code review, metrics/ measurements, architecture analysis, static analysis tools
Generic	follow best practices, training and experience, discuss with team, high quality aspirations, use version control

When looking at subgroups of developers based on gender, level of professional experience, or experience with responsibility for quality assurance, the rankings of properties show few differences (see Fig. 6). Most notably is that developers without professional experience in software development (N) rank *correctness* higher than the other groups (S and E). *Structure* and *readability* are ranked among the top-3 most important properties by all subgroups. The spearman rank correlations between groups are insignificant, except between the groups

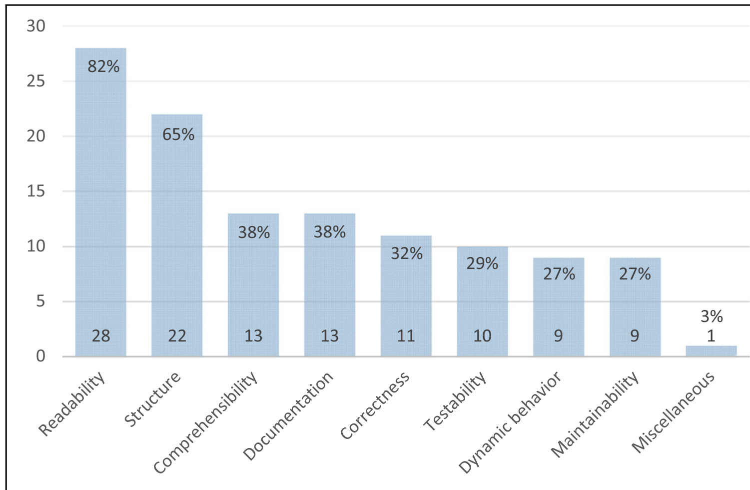


Fig. 3 Categories of terms used to define code quality (Q5). N=116, multiple mentionings were possible

without (N) and with extended (E) professional experience, which have a very high positive correlation (r_s=0.92, p<0.0001).

We also grouped the suggestions for quality improvement into analytical and constructive approaches as described in Section 3.5, see Fig. 7. Suggestions that could not be categorized as either analytical or constructive were categorized as *Other*. Of the 116 recommendations, 69% are constructive, 21.6% analytical and the remaining 8.6% other. The results show a significant difference between developers without and with responsibility for quality assurance, respectively ($\chi^2 = 11.76$, $p = .002792$) indicating that developers with responsibility for quality assurance have a stronger focus on analytical approaches.

> Answer to RQ1.2: *Structure*, *readability* and *documentation* are the most common target properties for quality improvement. Developers with responsibility for quality assurance have a significantly stronger focus on analytical approaches for quality improvement than developers without such responsibility.

Group	**Gender**		**Exp from prof sw dev**			**QA responsibility**	
	F	**M**	**N**	**S**	**E**	**N**	**Y**
Group size	6	28	12	11	11	23	11
Readability	1	1	1	1	1	1	1
Structure	2	2	2	2	2	2	2
Comprehensibility	7	3	3	4	4	4	2
Documentation	4	4	3	3	7	3	6
Correctness	4	5	6	7	2	8	2
Testability	7	5	8	4	4	6	5
Dynamic behavior	3	8	5	8	6	4	8
Maintainability	4	7	6	4	7	6	6
Miscellaneous	9	9	9	9	9	9	9

Fig. 4 Ranking of categories for defining properties of code quality (Q5) by groups of developers based on gender (**F**emale, **M**ale), professional experience (**N**o, **S**ome, **E**xtensive) and responsibility for quality assurance (**Y**es, **N**o)

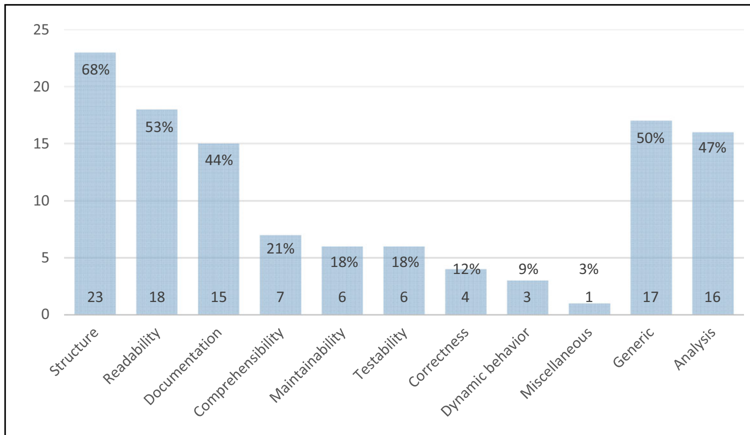


Fig. 5 Categories of terms used for suggestions to improve code quality (Q7). N=116, multiple mentionings were possible. (Note: Having N=116 for Q5 and for Q7 is a coincidence)

4.6 Properties Focused on in Code Examples (RQ2)

In interview question Q4, interviewees discussed their personal experiences using code they were asked to bring to the interview. This provided interviewees with opportunities for discussing aspects of code quality in detail from a personal point of view.

> *Q4*: We will now look at some of the code examples you provided. Please describe in detail, which properties or features you like or dislike with this code and how these properties or features affect the quality of the code. Please note that there are no correct or incorrect answers. We are primarily interested in code features that matter for you and why they do so.

During the coding phase, the 34 interviews were summarized into 721 “assertions” as described in Section 3.5. The overall results of this step are summarized in the heat map in Fig. 8, where the numbers in a cell *(row,column)* correspond to the numbers of assertions

Group	Gender		Exp from prof sw dev			QA responsibility	
	F	M	N	S	E	N	Y
Group size	6	28	12	11	11	23	11
Structure	1	1	1	2	1	1	2
Readability	3	2	3	3	3	2	2
Documentation	6	3	1	5	6	2	6
Comprehensibility	6	6	6	6	7	5	9
Maintainability	6	8	6	8	7	7	6
Testability	9	6	10	6	5	9	5
Correctness	4	10	4	10	10	7	10
Dynamic behavior	9	9	10	8	7	10	6
Miscellaneous	9	11	9	10	10	10	10
Generic	2	5	4	3	2	4	2
Analysis	4	3	8	1	4	6	1

Fig. 6 Ranking of categories for suggestions to improve code quality (Q7) by groups of developers based on gender (**F**emale, **M**ale), professional experience (**N**o, **S**ome, **E**xtensive) and responsibility for quality assurance (**N**o, **Y**es)

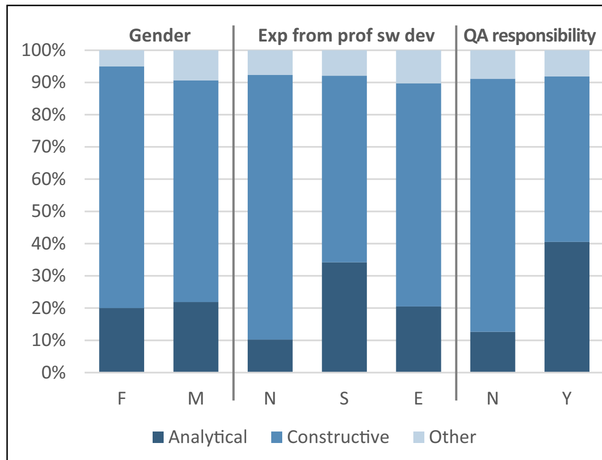


Fig. 7 Suggestions for type of quality improvement approach (Q7, analytical (bottom of column), constructive (middle) and other (top)) by groups of developers based on (from left to right) gender (**F**emale, **M**ale), professional experience (**N**o, **S**ome, **E**xtensive) and responsibility for quality assurance (**N**o, **Y**es)

row => *column*. E.g., the relationship *Documentation* => *Comprehensibility* appeared 61 times, i.e., 8.4% of all relationships.

Since the number of assertions varies substantially from interview to interview (min: 5, max: 54, avg: 21.2, median: 19), we normalized the results for each interview by dividing the number of occurrences of an assertion by the number of total assertions for the particular interview. As can be seen from Fig. 9, the normalized results are very similar to the non-normalized results in Fig. 8. The top-3- and the bottom-3-ranked categories are identical in both figures (see row/column RANK).

Figures 10, 11 and 12 show graphical overviews of the categories of the most frequently named properties in interviewees' assertions (normalized). The graphs show that *structure*, *comprehensibility*, *readability*, *documentation* and *maintainability* are consistently among the top-3 for all subgroups. The normalized heat-maps that are the basis for Figs. 10–12 can be found in Appendix C (Figs. 13, 14 and 15).

ALL	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	RANK
Comprehensibility	15	4	0	2	6	1	8	8	3	1	1	0	49	5
Correctness	17	7	0	4	6	0	9	3	2	9	0	0	57	4
Documentation	61	2	9	2	13	0	26	1	0	6	1	0	121	3
Dynamic behavior	3	1	0	5	2	0	2	0	0	2	0	0	15	9
Maintainability	7	3	0	3	16	1	2	6	1	2	0	0	41	6
Miscellaneous	4	5	0	3	3	0	5	0	0	1	0	0	21	8
Readability	40	6	2	2	12	0	72	9	5	12	2	0	162	2
Structure	52	4	2	15	47	0	31	38	10	12	1	0	212	1
Testability	2	7	2	6	9	0	1	1	8	0	0	0	36	7
Non-specific	0	0	0	0	0	0	0	0	0	0	0	0	0	12
Generic	2	0	1	0	0	0	1	0	0	1	0	0	5	10
Analysis	0	1	0	0	0	0	0	0	0	1	0	0	2	11
TOTAL	203	40	16	42	114	2	157	66	29	47	5	0		721
RANK	1	7	9	6	3	11	2	4	8	5	10	12	721	

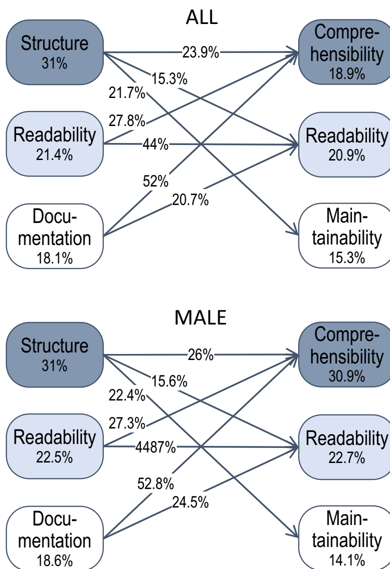
Fig. 8 Heat map for all 721 "assertions" about quality relationships in the 34 interviews (Q4). The number in cell (row, column) corresponds to the number of assertions *row* => *column*. E.g., assertion *documentation* => *comprehensibility* occurs 61 times, whereas *comprehensibility* => *documentation* occurs 0 times

ALL	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	RANK
Comprehensibility	0,67	0,18	0,00	0,06	0,39	0,07	0,35	0,40	0,08	0,07	0,06	0,00	2,35	4
Correctness	0,53	0,27	0,00	0,15	0,22	0,00	0,25	0,11	0,06	0,33	0,00	0,00	1,90	6
Documentation	3,19	0,08	0,52	0,06	0,56	0,00	1,27	0,13	0,00	0,26	0,06	0,00	6,14	3
Dynamic behavior	0,15	0,04	0,00	0,18	0,10	0,00	0,06	0,00	0,00	0,07	0,00	0,00	0,60	9
Maintainability	0,30	0,11	0,00	0,13	0,70	0,04	0,09	0,29	0,06	0,08	0,00	0,00	1,80	7
Miscellaneous	0,16	0,25	0,00	0,16	0,10	0,00	0,20	0,00	0,00	0,02	0,00	0,00	0,90	8
Readability	2,02	0,19	0,09	0,08	0,33	0,00	3,20	0,54	0,22	0,49	0,13	0,00	7,27	2
Structure	2,53	0,14	0,14	0,53	2,29	0,00	1,62	2,46	0,33	0,46	0,06	0,00	10,55	1
Testability	0,11	0,64	0,11	0,26	0,52	0,00	0,04	0,04	0,32	0,00	0,00	0,00	2,03	5
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	12
Generic	0,18	0,00	0,05	0,00	0,00	0,00	0,04	0,00	0,00	0,08	0,00	0,00	0,34	10
Analysis	0,00	0,06	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,06	0,00	0,00	0,12	11
TOTAL	9,84	1,95	0,92	1,62	5,20	0,11	7,11	3,96	1,07	1,92	0,31	0,00		34
RANK	1	5	9	7	3	11	2	4	8	6	10	12	34	

Fig. 9 Heat map for "assertions" about quality relationships after normalization (Q4)

The sources of the assertions for both gender subgroups look very similar. However, the female subgroup focuses more on *maintainability* and *structure* as targets of the assertions than the male group and less on *comprehensibility* and *readability*. Since our sample only comprised six females, it is difficult to draw general conclusions, though.

Regarding professional experience, we can see that the top-3 source properties are the same but decrease in weight successively with experience. For the subgroup with no experience, the top-3 categories account for 82% of the targets of all assertions, whereas they account for only 70.5% and 57.9%, respectively, for the subgroups with some and extensive experience. This indicates that more experienced developers have a broader picture of code quality. An interesting observation is that the subgroup with extensive experience ranks *readability* higher than *comprehensibility*. In this subgroup, we can also find a clear relationship between *readability* and *maintainability*, whereas we identified only a single assertion *readability* => *maintainability* in the other subgroups. Another observation is that the assertion



The left and right columns list the top-3 source and target properties, respectively, together with their relative frequency of all source and target properties.

Darker colors indicate higher relative frequencies.

Arrows indicate interviewees' "assertions" and their relative frequency, i.e. s —p→ t means that p% of all assertions with property s as the source have property t as the target.

In the figures, we only list assertion with p > 10%.

For easier cross-reference, the ordering of properties within columns is kept constant as far as possible.

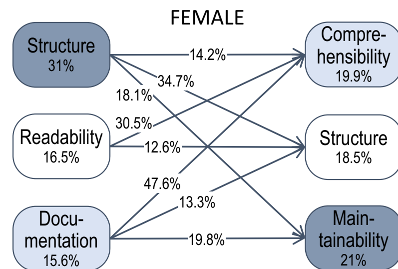


Fig. 10 Graphical overview of top-ranked categories for all interviewees (ALL), and by gender (MALE, n=28; FEMALE, n=6) for source- and target-properties and their relationships (Q4)

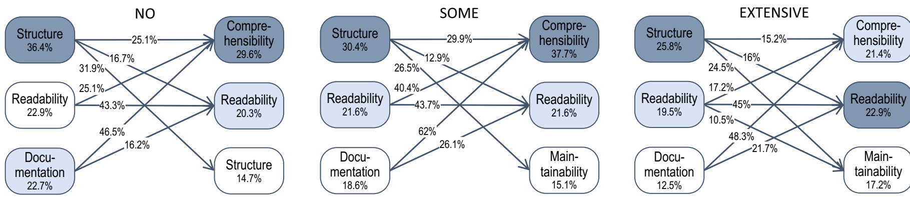


Fig. 11 Graphical overview of top-ranked categories grouped by interviewee experience (NO, n=11; SOME, n=11; EXTENSIVE, n=12) (Q4)

documentation => *comprehensibility* occurs much less frequent for developers with extensive experience. For this subgroup, *documentation* => *comprehensibility* accounts for 6% of all assertions, whereas it accounts for 10.6%–11.5% for the other experience subgroups and is the most frequent assertion for both of these groups (ignoring assertions where source and target belong to the same category).

A similar pattern can be observed between the subgroups with and without responsibility for quality assurance (see Fig. 12). Since 8 out of 11 QA responsibles also are in the subgroup with extensive experience (see Table 1), this observation could be related to overall experience.

When looking at individual labels (see Table 3), we found that 11.1% (80 of 721) of all labels for quality indicators dealt with comments/commenting, 7.9% (57 of 721) with naming and 6.8% (49 of 721) with formatting, indentation and whitespace. All other labels amounted to at most 4.3%.

Answer to RQ2: *Structure*, *comprehensibility*, *readability*, *documentation* and *maintainability* are the most commonly focused properties in developers' discussions about code examples. *Structure* is consistently mentioned by all groups as the most common source property leading to *comprehensibility*, *readability* and *maintainability*. The most commonly named code features are commenting, naming and formatting.

4.7 "Abstract" Perceptions of Code Quality Versus "Concrete" Manifestations in Code (RQ3)

In RQ3, we investigated in which ways developers' perceptions and discourses differ (or not) when talking about code quality in general (i.e., in an "abstract" way) and discussing

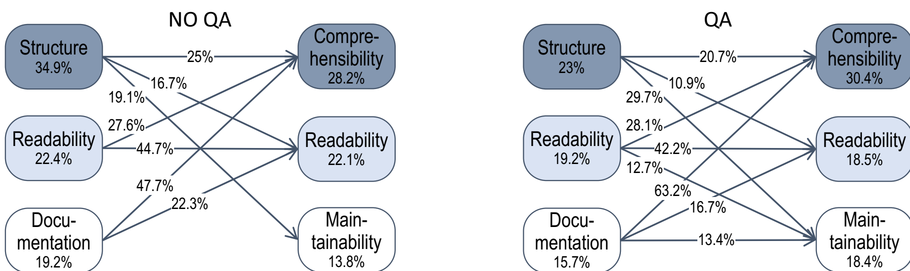


Fig. 12 Graphical overview of top-ranked categories grouped by responsibility for quality assurance (NO QA, n=23; QA, n=11) (Q4)

the quality of specific code examples and the trade-offs that might manifest in those. To answer RQ3, we compared the results regarding code quality categories from results in RQ1 (based on an abstract view of the interviewees; Q5–Q8) with the results from RQ2 (based on the specific code examples provided by the interviewees; Q4). The results are not directly comparable as the answers to RQ1 are based on frequencies of occurrences of code quality aspects, whereas the answers to RQ2 also consider relationships between aspects of quality. Still, the ranks of the categories in each of the results can be used as importance indicators for the code quality categories. Thus, a qualitative comparison of the results is possible.

When defining code quality, all groups of developers consistently use *readability* and *structure* as the first and second most frequently named property (see RQ1.1, Section 4.4). Interestingly, developers with extensive experience and those with QA responsibilities rated *correctness* much higher than all other groups while they considered *documentation* much less relevant. The suggestions for quality improvements overall are less consistent but focus mainly on *structure*, *readability* and *documentation* (see RQ1.2, Section 4.5). Again, we can see that less experienced developers and those without QA responsibilities consider *documentation* more important than developers with extensive experience.

When discussing the specific code examples the interviewees brought along, they focused on *structure*, *readability* and *documentation*, primarily as sources in the assertions. However, the discussions showed that these properties were not seen as the ultimate quality goals but rather properties that lead to *comprehensibility* and *maintainability* (as well as *readability* and *structure*, see Figs. 10–12). Interestingly, not a single developer without professional experience mentioned *comprehensibility* as a source in an assertion (see Fig. 14, at top, first row), whereas it ranks fourth overall (see Fig. 9). Yet, they rather considered it a consequence of other properties (see Fig. 14, at top, first column).

Comparing these results to answer RQ3, there are quality categories that are considered important on both the abstract and the concrete level, such as *structure*, *readability* and *documentation*. However, the developers with extensive experience and those with QA responsibility ranked *documentation* much lower on the abstract level than they did on the concrete examples.

Due to their high experience, they might require less documentation and therefore consider it less important in the abstract view than less experienced developers.

Less experienced developers, on the other hand, did not mention *correctness* as often on the abstract level as they did on the concrete examples. The code examples might have helped them recognize the importance of code correctness, whereas experienced developers immediately considered this an important aspect.

The previous findings are also reflected in citations from the interviews, e.g., reasoning about *readability*:

> *"Readability! Team projects! Basically, is the only reason. If you are writing code for yourself doing something only you need and you can do it in a day and you don't need to return to the code, which is usually [what you] tell to yourself, but you do need to return."* (Interviewee RD1)

> *"You can often further simplify code, optimize, and that can sometimes go on until you reach a point where it compiles undoubtedly very quickly, but where, as a human being, you cannot read it easily anymore. So I think it is important that it still is a bit readable."* (Interviewee JJ3)

Interviewee HK2, for example, explicitly emphasized *readability* over other quality attributes when comparing code:

"I think I would go for ... because it's such a simple piece of code, with such low cyclomatic complexity, so few different paths, that I think readability is more important in this case than robustness and testability."(Interviewee HK2)

Several interviewees use terms like simplicity or elegance to describe *comprehensibility*:

"Describing things as plain, as simple, as well structured as possible. Focus on the real content as far as possible. ... Understandability doesn't mean that is necessarily terse, but it means, you look at it, and you understand very quickly what it is doing." (Interviewee HS3)

"There are many things that can be done with very elegant one-liners, that are still understandable." (Interviewee RD4)

Answer to RQ3: *Comprehensibility* and *maintainability* become more important when interviewees discuss concrete code examples. The abstract properties *structure*, *readability* and *documentation* are still seen as important but mainly as a prerequisite for *comprehensibility* and *maintainability*.

5 Discussion

The most important and obvious results from the RQs examined above are that developers strive for comprehensible and maintainable code. These qualities are achieved by structured, readable and documented code, especially when looking at specific code examples. These findings are in line with some of the results in the related work (see Section 2). This means that professionally developed code should be well structured, readable and appropriately documented. These aspects should be emphasized in programming and software education to prepare students for their professional careers.

Lesson learned 1: Developers strive for *comprehensibility* and *maintainability*, which means that code should be well structured, readable and appropriately documented. To prepare students for their professional careers, these aspects should be emphasized in programming education.

Regarding what appropriately documented code means, responses were diverse. Although comments/commenting was the most frequent label for quality indicators, interviewees did not generally call for more documentation. On the one hand, comments are seen as important, e.g.,

"The other thing is comments. They are important. I find that I actually write code better if I am commenting it, so if I were writing a SQL statement for someone else that I have explain to, and they need to understand what I am doing, and I actually write out the comments for it, then I actually write the code better." (Interviewee SH1)

On the other hand, they can also impede comprehension. For simple code

"...you actually get [a] better idea without reading the comments because when you read a comment it breaks your idea of what is happening actually in the code." (Interviewee RD2)

Some interviewees even called for self-documenting code by using appropriate method names instead of using commenting

"And instead of adding comments here... I would consider it more reasonable to extract methods here that get meaningful names, so that you could read the code based on the method names..." (Interviewee CK2)

Finally, commented-out code is described as an issue by several interviewees, e.g.,

"It makes the whole a lot less clear, also there are all these comments, which distracts. ... it is all commented out code. It is just code from which they thought they would later reuse it, but never removed it. I think this creates quite a lot of confusion." (Interviewee JA3)

Summarizing the findings on documentation of code, we can say that while documented code is considered important, that may not necessarily mean that adding more comments is always good. Better comprehensibility might also be achieved by meaningful naming and adequate structuring.

Lesson learned 2: While commenting code is considered important, comments may also be a distraction. Meaningful names and adequate structuring might improve *comprehensibility* more.

This corroborates the results of Plösch et al. (2014)'s survey on software documentation results, which showed that people working with software development projects consider accuracy, clarity, readability, structuredness, and understandability as especially important for software documentation.

The issue with documentation is complemented by the results across the different experience groups. While developers without professional experience rated documentation rather high to improve code quality, experienced and QA-responsible developers rated it much lower. A possible conclusion might be that while explicit documentation seems to improve aspects like comprehensibility and maintainability initially, this effect may decrease with experience. The highly experienced group rated correctness of code more important for high quality. However, focusing only on correctness might make it more difficult to onboard less experienced developers.

Another aspect that evolved from the interviews, is concerned with the level of measuring code quality. While some interviewees were concerned with micro-level quality aspects such as lines of comments, variable naming or code formatting (among others), others focused more on higher level quality aspects, like overall structure. These range from encapsulation and providing a specific, single functionality in one part of the code to the overall structure of a software project such as low coupling and decomposition.

"But, of course, you see if you look into it more deeply, that the structure is completely different...I basically do not have a real application layer there." (Interviewee CK1)

We can conclude that code quality has to be assessed and maintained on several levels. Assuring high quality on one level does not guarantee high quality on others, e.g., nicely formatted and named code which is correct may still be arranged inappropriately within the whole software system. Similarly, a nicely decomposed and layered software architecture

does not assure comprehensible and maintainable code within a single unit. Thus code quality is a complex and diverse property to achieve for a software project.

Lesson learned 3: Code quality is a complex property that should be assessed and maintained on several levels. Assuring high quality on the micro-level (e.g., statements) does not guarantee high quality on system level.

Another interesting observation is that testability of code has not been rated as high as one might have expected. More experienced developers and those with QA-responsibility mentioned this slightly more than others. Yet, it still remains in the middle to lower ranges in all groups. This also holds true when looking at the abstract perception level.

Miguel et al. (2014) reviewed common software quality models and provide a list of 50+ terms used in those models to describe qualities together with their definitions. This list only captures maintainability with a definition that is in line with our results. Furthermore, comprehensibility is captured as "appropriateness recognizability" (with the synonym "understandability"). Structure, readability and documentation are not covered. One reason might be that common software quality standards/models target software systems as products rather than the code these systems comprise. This may make it difficult for developers to relate their work to those standards in practice (Al-Kilidar et al. 2005).

6 Summary and Conclusions

In this study, we investigated how software developers perceive or understand source code quality. Overall, we analyzed and discussed data from 34 semi-structured interviews with software developers from Europe and the US. To facilitate the discussion of first-hand experiences of code quality, study participants brought along their own code examples.

Perceptions of code quality were grouped into eleven categories or themes: readability, structure, comprehensibility, maintainability, correctness, documentation, dynamic behavior, testability, correctness, maintainability, analysis, generic and miscellaneous. Since there are no empirically based categories for (developers' perceptions of) code quality, our categories could be a good starting point for other researchers who want to study perceptions of code quality.

Our results show that developers consistently focus on structure, readability and documentation as important qualities of code to make it comprehensible and maintainable *for other developers*. In doing so, developers establish themselves as a group of stakeholders that might need to be considered more thoroughly in common software quality models or standards. Common quality models or standards focus on "product quality" and "quality in use" (as, e.g., ISO 25010 ISO/IEC 2011) targeting the customers or end users of software. This issue has also been noted by a study group on the ISO/IEC 25000 family of software quality standards (also known as SQuaRE) (Nakajima 2019, p. 1): "Most of the measures in ISO/IEC 25023 are quality-in-use measures since the specified measures are about external properties at runtime. There are several coding standards such as MISRA, AUTOSAR, and CISQ, which provide the checklists or rules for code to entail quality measures. For SQuaRE to be considered a strong guide for measurement of software and systems product quality, it must improve how it guides for quality measurement of internal properties."

More research is therefore necessary to (a) define indicators that capture code properties that matter for developers, to (b) develop software quality models that are relevant for developers and to (c) define metrics that reliably measure the quality attributes that are relevant for developers. Metrics for, e.g., readability (Buse and Weimer 2010) and comprehensibility (Scalabrino et al. 2019) have been defined and studied before. However, we see a need for further empirical research in professional software development contexts.

Appendix A: Interview Questions

ID	Question text	Answer format
Q1	Gender?	M/F
Q2	In which country did you get most of your experience?	Short text
Q3-1	What is your current occupation and job title (if applicable)?	Student, professional programmer, educator
Q3-2	How many years of experience as a professional programmer do you have and how recent is that experience?	Number
Q3-3	How many years of teaching software development do you have, in terms of full-time years?	Number
Q3-S1	What is your study program and level (e.g., Bachelor of Software Engineering)?	Short text
Q3-S2	Which study year are you in, in terms of full-time study equivalents?	Number
Q3-S3	Is programming one of your main study subjects?	Y/N
Q3-S4	How many programming courses did you take, i.e. courses with a significant programming component?	Number
Q3-S5	How many programming languages can you program in?	List of names
Q3-S6	How large was the largest program you developed?	Number
Q3-P1	What is your job title?	Short text
Q3-P2	Do your formal responsibilities involve quality assurance?	Y/N
Q3-P3	Which programming languages do you use most?	List of names
Q3-E1	How many courses related to programming do you teach per year on average?	Number
Q3-E2	Which courses related to programming did you teach during the last five years?	List of names
Q3-E3	Do you talk about code quality in those courses?	Y/N
Q3-E4	If yes, in which courses?	List of names
Q3-4	Which programming languages do you prefer most?	List of names
Q3-5	On a scale from strongly disagree to strongly agree, how much do you agree or disagree with the following statements regarding your personal experience related to software development?	
Q3-5a	I read and modify source code from other programmers.	Likert-type scale
Q3-5b	Other people are reading and modifying the code that I write.	Likert-type scale
Q3-5c	I review or comment other people's code.	Likert-type scale

ID	Question text	Answer format
Q3-5d	Other people review or comment the code that I write.	Likert-type scale
Q4	We will now look at some of the code examples you provided. Please describe in detail, which properties or features you like or dislike with this code and how these properties or features affect the quality of the code. Please note that there are no correct or incorrect answers. We are primarily interested in code features that matter for you and why they do so.	Audio-recorded and transcribed
Q5	How would you define code quality? Which properties, features or indicator show you, personally, something about quality?	Text
Q6	On a scale from strongly disagree to strongly agree, how much do you agree or disagree with the following statements regarding your personal experience related to source code quality?	
Q6a	Code quality is of high importance in my work/studies/teaching.	Likert-type scale
Q6b	I can easily tell good from bad code.	Likert-type scale
Q6c	I regularly work with code quality issues.	Likert-type scale
Q6d	I know how to measure code quality.	Likert-type scale
Q6e	I have learned a lot about code quality during my education.	Likert-type scale
Q6f	I have learned a lot about code quality from my colleagues.	Likert-type scale
Q6g	I have learned a lot about code quality from the Internet.	Likert-type scale
Q7	Please provide your top-3 recommendations for increasing the quality of code. Please indicate when a recommendation applies in special cases only.	
Q7-1	My top recommendation for achieving high code quality.	Short text
Q7-2	My second most important recommendation for achieving high code quality.	Short text
Q7-3	My third most important recommendation for achieving high code quality.	Short text
Q7-4	Any further important recommendations you want to mention?	Short text
Q8	According to your experience, what are the three topmost quality factors or indicators of high quality code?	
Q8-1	The most important quality factor/indicator for high quality code.	Short text
Q8-2	The second most important quality factor/indicator for high quality code.	Short text
Q8-3	The third most important quality factor/indicator for high quality code.	Short text
Q8-4	Any further highly important factors you want to mention?	Short text
Q9	According to your experience, what are the three most useful sources of information about code quality? Are these sources reliable and trustworthy?	
Q9-1	The most useful source of information about software quality.	Short text
Q9-2	The second most useful source of information about software quality.	Short text
Q9-3	The third most useful source of information about software quality.	Short text
Q9-4	Any further highly useful resources you want to mention?	Short text
Q10	According to your experience, what are the three most useful tools for improving code quality or achieving high quality code?	

ID	Question text	Answer format
Q10-1	The most useful tool for improving code quality.	Short text
Q10-2	The second most useful tool for improving code quality.	Short text
Q10-3	The third most useful tool for improving code quality.	Short text
Q10-4	Any further highly useful tools you want to mention?	Short text
Q11	Is there anything more you would like to bring up?	Text

Appendix B: Supplementary Materials

B.1 Study Design Package

To ensure that all researchers follow the same procedures, two of the co-authors developed a study design package with common guidelines, instructions and templates including the following:

- a study guide describing the overall study design;
- a detailed interview script with instructions regarding the phrasing of interview questions including suggestions for probing questions;
- a participant information sheet to ensure that all participants receive the same information about the study and the example code they should bring along;
- a consent form that participants need to sign;
- a transcription guide with guidelines for transcribing the interviews; and
- an analysis guide for the coding of Q4.

The complete study design package can be downloaded at https://10.5281/zenodo.8233989.

B.2 Data

The raw data for questions Q1–Q3 & Q5–Q11 and the assertions for question Q4 can be downloaded at https://10.5281/zenodo.8233989.

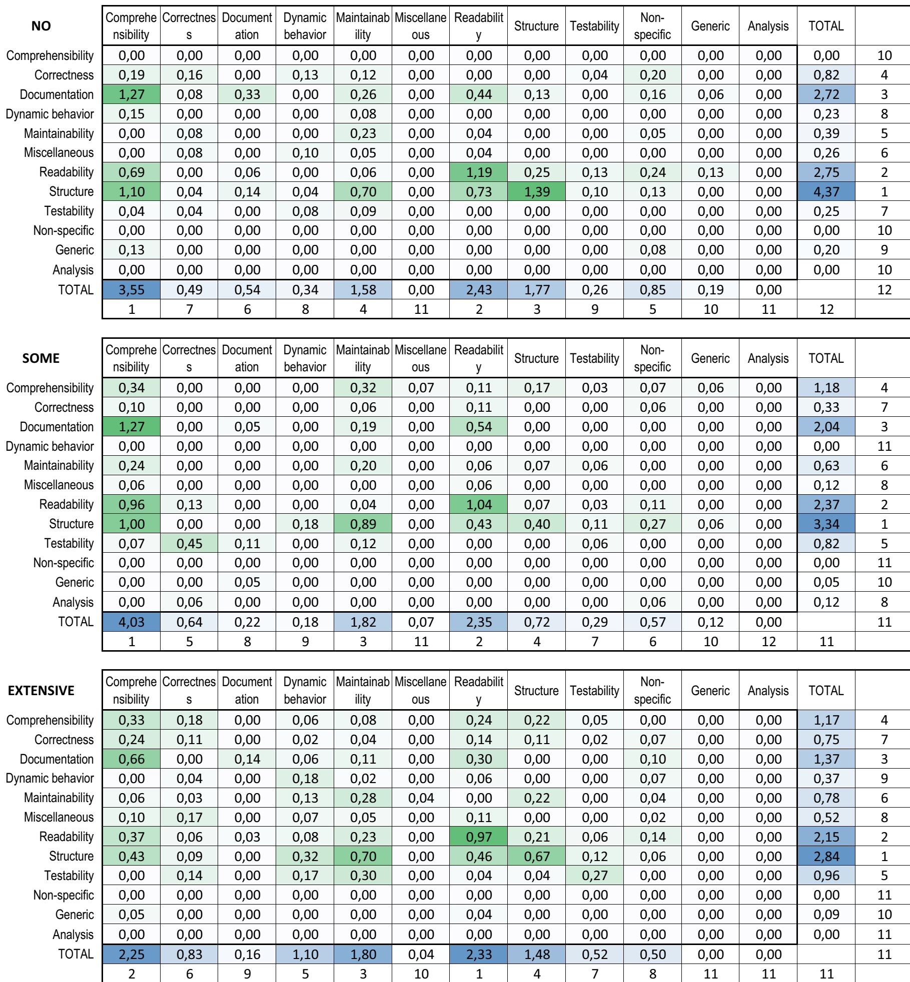
Appendix C: Heat Maps

Normalized heat-maps for all subgroups of developers based on gender (see Fig. 13), level of professional experience (see Fig. 14) and experience with responsibility for quality assurance (see Fig. 15).

MALE	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	
Comprehensibility	0,62	0,18	0,00	0,06	0,34	0,07	0,35	0,40	0,08	0,07	0,00	0,00	2,17	4
Correctness	0,53	0,14	0,00	0,06	0,12	0,00	0,19	0,06	0,06	0,22	0,00	0,00	1,38	6
Documentation	2,75	0,04	0,48	0,06	0,37	0,00	1,27	0,00	0,00	0,17	0,06	0,00	5,20	3
Dynamic behavior	0,15	0,04	0,00	0,18	0,10	0,00	0,06	0,00	0,00	0,07	0,00	0,00	0,60	9
Maintainability	0,30	0,06	0,00	0,00	0,31	0,04	0,04	0,16	0,06	0,04	0,00	0,00	0,99	7
Miscellaneous	0,16	0,20	0,00	0,07	0,05	0,00	0,20	0,00	0,00	0,02	0,00	0,00	0,71	8
Readability	1,72	0,19	0,09	0,08	0,33	0,00	2,82	0,41	0,22	0,31	0,13	0,00	6,29	2
Structure	2,26	0,09	0,14	0,49	1,95	0,00	1,36	1,82	0,33	0,24	0,00	0,00	8,69	1
Testability	0,11	0,59	0,11	0,17	0,38	0,00	0,04	0,00	0,24	0,00	0,00	0,00	1,64	5
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	12
Generic	0,05	0,00	0,05	0,00	0,00	0,00	0,04	0,00	0,00	0,08	0,00	0,00	0,22	10
Analysis	0,00	0,06	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,06	0,00	0,00	0,12	11
TOTAL	8,64	1,61	0,87	1,17	3,94	0,11	6,36	2,85	0,99	1,27	0,19	0,00		28
	1	5	9	7	3	11	2	4	8	6	10	12	28	

FEMALE	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	
Comprehensibility	0,06	0,00	0,00	0,00	0,06	0,00	0,00	0,00	0,00	0,00	0,06	0,00	0,18	8
Correctness	0,00	0,13	0,00	0,09	0,10	0,00	0,06	0,04	0,00	0,11	0,00	0,00	0,52	5
Documentation	0,45	0,04	0,04	0,00	0,19	0,00	0,00	0,13	0,00	0,10	0,00	0,00	0,94	3
Dynamic behavior	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	10
Maintainability	0,00	0,05	0,00	0,13	0,40	0,00	0,06	0,13	0,00	0,05	0,00	0,00	0,81	4
Miscellaneous	0,00	0,05	0,00	0,10	0,05	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,19	7
Readability	0,30	0,00	0,00	0,00	0,00	0,00	0,38	0,13	0,00	0,18	0,00	0,00	0,99	2
Structure	0,26	0,04	0,00	0,04	0,34	0,00	0,26	0,64	0,00	0,21	0,06	0,00	1,86	1
Testability	0,00	0,04	0,00	0,08	0,13	0,00	0,00	0,04	0,09	0,00	0,00	0,00	0,39	6
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	10
Generic	0,13	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,13	9
Analysis	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	10
TOTAL	1,20	0,35	0,04	0,44	1,26	0,00	0,76	1,11	0,09	0,64	0,12	0,00		6
	2	7	10	6	1	11	4	3	9	5	8	11	6	

Fig. 13 Normalized heat maps for all "assertions" (Q4) grouped by gender (male at the top)



NO	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	
Comprehensibility	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	10
Correctness	0,19	0,16	0,00	0,13	0,12	0,00	0,00	0,00	0,04	0,20	0,00	0,00	0,82	4
Documentation	1,27	0,08	0,33	0,00	0,26	0,00	0,44	0,13	0,00	0,16	0,06	0,00	2,72	3
Dynamic behavior	0,15	0,00	0,00	0,00	0,08	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,23	8
Maintainability	0,00	0,08	0,00	0,00	0,23	0,00	0,04	0,00	0,00	0,05	0,00	0,00	0,39	5
Miscellaneous	0,00	0,08	0,00	0,10	0,05	0,00	0,04	0,00	0,00	0,00	0,00	0,00	0,26	6
Readability	0,69	0,00	0,06	0,00	0,06	0,00	1,19	0,25	0,13	0,24	0,13	0,00	2,75	2
Structure	1,10	0,04	0,14	0,04	0,70	0,00	0,73	1,39	0,10	0,13	0,00	0,00	4,37	1
Testability	0,04	0,04	0,00	0,08	0,09	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,25	7
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	10
Generic	0,13	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,08	0,00	0,00	0,20	9
Analysis	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	10
TOTAL	3,55	0,49	0,54	0,34	1,58	0,00	2,43	1,77	0,26	0,85	0,19	0,00		12
	1	7	6	8	4	11	2	3	9	5	10	11	12	

SOME	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	
Comprehensibility	0,34	0,00	0,00	0,00	0,32	0,07	0,11	0,17	0,03	0,07	0,06	0,00	1,18	4
Correctness	0,10	0,00	0,00	0,00	0,06	0,00	0,11	0,00	0,00	0,06	0,00	0,00	0,33	7
Documentation	1,27	0,00	0,05	0,00	0,19	0,00	0,54	0,00	0,00	0,00	0,00	0,00	2,04	3
Dynamic behavior	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	11
Maintainability	0,24	0,00	0,00	0,00	0,20	0,00	0,06	0,07	0,06	0,00	0,00	0,00	0,63	6
Miscellaneous	0,06	0,00	0,00	0,00	0,00	0,00	0,06	0,00	0,00	0,00	0,00	0,00	0,12	8
Readability	0,96	0,13	0,00	0,00	0,04	0,00	1,04	0,07	0,03	0,11	0,00	0,00	2,37	2
Structure	1,00	0,00	0,00	0,18	0,89	0,00	0,43	0,40	0,11	0,27	0,06	0,00	3,34	1
Testability	0,07	0,45	0,11	0,00	0,12	0,00	0,00	0,00	0,06	0,00	0,00	0,00	0,82	5
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	11
Generic	0,00	0,00	0,05	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,05	10
Analysis	0,00	0,06	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,06	0,00	0,00	0,12	8
TOTAL	4,03	0,64	0,22	0,18	1,82	0,07	2,35	0,72	0,29	0,57	0,12	0,00		11
	1	5	8	9	3	11	2	4	7	6	10	12	11	

EXTENSIVE	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	
Comprehensibility	0,33	0,18	0,00	0,06	0,08	0,00	0,24	0,22	0,05	0,00	0,00	0,00	1,17	4
Correctness	0,24	0,11	0,00	0,02	0,04	0,00	0,14	0,11	0,02	0,07	0,00	0,00	0,75	7
Documentation	0,66	0,00	0,14	0,06	0,11	0,00	0,30	0,00	0,00	0,10	0,00	0,00	1,37	3
Dynamic behavior	0,00	0,04	0,00	0,18	0,02	0,00	0,06	0,00	0,00	0,07	0,00	0,00	0,37	9
Maintainability	0,06	0,03	0,00	0,13	0,28	0,04	0,00	0,22	0,00	0,04	0,00	0,00	0,78	6
Miscellaneous	0,10	0,17	0,00	0,07	0,05	0,00	0,11	0,00	0,00	0,02	0,00	0,00	0,52	8
Readability	0,37	0,06	0,03	0,08	0,23	0,00	0,97	0,21	0,06	0,14	0,00	0,00	2,15	2
Structure	0,43	0,09	0,00	0,32	0,70	0,00	0,46	0,67	0,12	0,06	0,00	0,00	2,84	1
Testability	0,00	0,14	0,00	0,17	0,30	0,00	0,04	0,04	0,27	0,00	0,00	0,00	0,96	5
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	11
Generic	0,05	0,00	0,00	0,00	0,00	0,00	0,04	0,00	0,00	0,00	0,00	0,00	0,09	10
Analysis	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	11
TOTAL	2,25	0,83	0,16	1,10	1,80	0,04	2,33	1,48	0,52	0,50	0,00	0,00		11
	2	6	9	5	3	10	1	4	7	8	11	11	11	

Fig. 14 Normalized heat maps for all "assertions" (Q4) grouped by professional experience (no experience at the top, some experience in the middle, extensive experience at the bottom)

NO QA	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	
Comprehensibility	0,22	0,00	0,00	0,03	0,06	0,07	0,14	0,21	0,03	0,07	0,06	0,00	0,89	7
Correctness	0,19	0,21	0,00	0,13	0,17	0,00	0,11	0,04	0,04	0,29	0,00	0,00	1,17	5
Documentation	2,11	0,08	0,47	0,00	0,32	0,00	0,99	0,13	0,00	0,26	0,06	0,00	4,42	3
Dynamic behavior	0,15	0,04	0,00	0,09	0,08	0,00	0,00	0,00	0,00	0,03	0,00	0,00	0,39	9
Maintainability	0,11	0,08	0,00	0,13	0,60	0,00	0,09	0,25	0,06	0,05	0,00	0,00	1,37	4
Miscellaneous	0,06	0,08	0,00	0,10	0,08	0,00	0,09	0,00	0,00	0,00	0,00	0,00	0,41	8
Readability	1,43	0,13	0,06	0,08	0,06	0,00	2,31	0,36	0,16	0,45	0,13	0,00	5,16	2
Structure	2,00	0,09	0,14	0,33	1,54	0,00	1,34	1,88	0,21	0,44	0,06	0,00	8,02	1
Testability	0,11	0,09	0,05	0,26	0,26	0,00	0,00	0,04	0,09	0,00	0,00	0,00	0,90	6
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	11
Generic	0,13	0,00	0,05	0,00	0,00	0,00	0,00	0,00	0,00	0,08	0,00	0,00	0,25	10
Analysis	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	11
TOTAL	6,49	0,80	0,78	1,14	3,17	0,07	5,07	2,91	0,58	1,67	0,31	0,00		23
	1	7	8	6	3	11	2	4	9	5	10	12	23	

QA	Comprehensibility	Correctness	Documentation	Dynamic behavior	Maintainability	Miscellaneous	Readability	Structure	Testability	Non-specific	Generic	Analysis	TOTAL	
Comprehensibility	0,46	0,18	0,00	0,03	0,34	0,00	0,21	0,19	0,05	0,00	0,00	0,00	1,45	4
Correctness	0,34	0,06	0,00	0,02	0,04	0,00	0,14	0,06	0,02	0,04	0,00	0,00	0,73	6
Documentation	1,09	0,00	0,05	0,06	0,23	0,00	0,29	0,00	0,00	0,00	0,00	0,00	1,72	3
Dynamic behavior	0,00	0,00	0,00	0,10	0,02	0,00	0,06	0,00	0,00	0,03	0,00	0,00	0,21	9
Maintainability	0,19	0,03	0,00	0,00	0,10	0,04	0,00	0,04	0,00	0,04	0,00	0,00	0,43	8
Miscellaneous	0,10	0,17	0,00	0,07	0,02	0,00	0,11	0,00	0,00	0,02	0,00	0,00	0,48	7
Readability	0,59	0,06	0,03	0,00	0,27	0,00	0,89	0,17	0,06	0,04	0,00	0,00	2,12	2
Structure	0,52	0,05	0,00	0,20	0,75	0,00	0,27	0,58	0,12	0,02	0,00	0,00	2,53	1
Testability	0,00	0,54	0,06	0,00	0,26	0,00	0,04	0,00	0,24	0,00	0,00	0,00	1,13	5
Non-specific	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	12
Generic	0,05	0,00	0,00	0,00	0,00	0,00	0,04	0,00	0,00	0,00	0,00	0,00	0,09	11
Analysis	0,00	0,06	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,06	0,00	0,00	0,12	10
TOTAL	3,34	1,15	0,14	0,48	2,02	0,04	2,04	1,05	0,49	0,24	0,00	0,00		11
	1	4	9	7	3	10	2	5	6	8	11	11	11	

Fig. 15 Normalized heat maps for all "assertions" (Q4) grouped by responsibility for quality assurance (No responsibility at the top)

Acknowledgements This work has in parts been supported by ELLIIT; the Swedish Strategic Research Area in IT and Mobile Communications. The authors thank all interviewees for sharing their examples and experience.

Author Contributions Jürgen Börstler and Daniel Toll contributed to the study conception, design and material preparation. Data collection and preparation were performed by Jürgen Börstler, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar, Harald Störrle, Daniel Toll and Jelle van Assema. Data analysis was performed by Kwabena Ebo Bennin, Jürgen Börstler, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner and Bonnie MacKellar. The first draft of the manuscript was written by Kwabena Ebo Bennin and Jürgen Börstler and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Open access funding provided by Blekinge Institute of Technology.

Declarations

Conflicts of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Ajami S, Woodbridge Y, Feitelson DG (2017) Syntax, predicates, idioms: what really affects code complexity? In: Proceedings of the 25th international conference on program comprehension, pp 66–76
Al Dallal J, Abdin A (2018) Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: a systematic literature review. IEEE Trans Softw Eng 44(1):44–69
Al-Kilidar H, Cox K, Kitchenham B (2005) The use and usefulness of the iso/iec 9126 quality standard. In: Proceedings of the 4th international symposium on empirical software engineering, pp 126–132
Alshayeb M (2009) Empirical investigation of refactoring effect on software quality. Inf Softw Technol 51(9):1319–1326
Antinyan V, Staron M, Sandberg A (2017) Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. Empir Softw Eng 22(6):3057–3087
Avidan E, Feitelson DG (2017) Effects of variable names on comprehension an empirical study. In: Proceedings of the 25th international conference on program Comprehension, pp 55–65
Bakota T, Hegedűs P, Körtvélyesi P, Ferenc R, Gyimóthy T (2011) A probabilistic software quality model. In: Proceedings of the 27th IEEE international conference on software maintenance, pp 243–252
Bavota G, Russo B (2015) Four eyes are better than two: On the impact of code reviews on software quality. In: Proceedings of the 31st IEEE international conference on software maintenance and evolution, pp 81–90
Beniamini G, Gingichashvili S, Orbach AK, Feitelson DG (2017) Meaningful identifier names: the case of single-letter variables. In: Proceedings of the 25th international conference on program comprehension, pp 45–54
Börstler J, Störrle H, Toll D, van Assema J, Duran R, Hooshangi S, Jeuring J, Keuning H, Kleiner C, MacKellar B (2018) "I know it when I see it" – Perceptions of code quality: ITiCSE'17 Working Group Report. In: Proceedings of the 2017 ITiCSE conference working group reports, pp 70–85
Buse RP, Weimer WR (2010) Learning a metric for code readability. IEEE Trans Softw Eng 36(4):546–558
Butler S, Wermelinger M, Yu Y, Sharp H (2010) Exploring the influence of identifier names on code quality: An empirical study. In: Proceedings of the 14th european conference on software maintenance and reengineering, pp 156–165
Chatzigeorgiou A, Manakos A (2010) Investigating the evolution of bad smells in object-oriented code. In: 2010 seventh international conference on the quality of information and communications technology, IEEE, pp 106–115
Correia JP, Kanellopoulos Y, Visser J (2009) A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics. In: Proceedings of the 27th IEEE international conference on software maintenan, pp 61–70
Creswell JW, Creswell JD (2017) Research design: Qualitative, quantitative, and mixed methods approaches. Sage publications
Curtis B, Martin RA, Douziech PE (2022) Measuring the structural quality of software systems. Computer 55(3):87–90
Davila N, Nunes I (2021) A systematic literature review and taxonomy of modern code review. J Syst Softw 177:110951
Fakhoury S, Ma Y, Arnaoudova V, Adesope O (2018) The effect of poor source code lexicon and readability on developers' cognitive load. In: Proceedings of the 26th conference on program comprehension, pp 286–296
Fenton N, Bieman J (2014) Software metrics: a rigorous and practical approach. CRC Press
Freelon D (2013) Recal oir: ordinal, interval, and ratio intercoder reliability as a web service. International Journal of Internet Science 8(1):10–16
Gil Y, Lalouche G (2017) On the correlation between size and metric validity. Empir Softw Eng 22(5):2585–2611
Gorla N, Lin SC (2010) Determinants of software quality: A survey of information systems project managers. Inf Softw Technol 52(6):602–610
Green R, Ledgard H (2011) Coding guidelines: Finding the art in the science. Communications of the ACM 54(12):57–63
Hatton L (2008) Testing the value of checklists in code inspections. IEEE Software 25(4):82–88
Hennink M, Kaiser BN (2022) Sample sizes for saturation in qualitative research: A systematic review of empirical tests. Social Science & Medicine 292:114523
Humble Á (2015) Guide to transcribing. https://www.msvu.ca/wp-content/uploads/2020/05/GuideTranscribing.pdf, Accessed 29 Nov 2021

ISO/IEC (2011) Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – system and software quality models. Tech. Rep. ISO/IEC 25010:2011, International Organization for Standardization, Geneva, Switzerland
Jabangwe R, Börstler J, Šmite D, Wohlin C (2015) Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. Empir Softw Eng 20(3):640–693
Jørgensen M (1999) Software quality measurement. Adv Eng Softw 30(12):907–912
Kim M, Zimmermann T, Nagappan N (2014) An empirical study of refactoring challenges and benefits at Microsoft. IEEE Transactions on Software Engineering 40(7):633–649
Ko AJ, Myers BA (2005) A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages & Computing 16(1–2):41–84
Kollanus S, Koskinen J (2009) Survey of software inspection research. The Open Software Engineering Journal 3(1):15–34
Kononenko O, Baysal O, Godfrey MW (2016) Code review quality: How developers see it. In: Proceedings of the 38th international conference on software engineering, pp 1028–1038
Malaquias R, Ribeiro M, Bonifácio R, Monteiro E, Medeiros F, Garcia A, Gheyi R (2017) The discipline of preprocessor-based annotations – does #ifdef tag n't #endif matter. In: Proceedings of the 25th international conference on program comprehension, pp 297–307
Martin RC (2008) Clean code: a handbook of agile software craftsmanship. Prentice-Hall
Menzies T, Greenwald J, Frank A (2006) Data mining static code attributes to learn defect predictors. Trans Softw Eng 33(1):2–13
Miguel JP, Mauricio D, Rodríguez G (2014) A review of software quality models for the evaluation of software products. International Journal of Software Engineering & Applications 5(6):31–53
Müller SC, Fritz T (2016) Using (bio) metrics to predict code quality online. In: Proceedings of the 38th international conference on software engineering, pp 452–463
Nakajima T (2019) Study group report on SQuaRE future direction. In: Proceedings of the 1st international workshop on experience with square series and its future direction (co-located with 26th Asia-Pacific Software Engineering Conference), pp 1–5
Nistala P, Nori KV, Reddy R (2019) Software quality models: A systematic mapping study. In: Proceedings of the 13th IEEE/ACM international conference on software and system processes, pp 125–134
Nuñez-Varela AS, Pérez-Gonzalez HG, Martínez-Perez FE, Soubervielle-Montalvo C (2017) Source code metrics: A systematic mapping study. J Syst Softw 128:164–197
Pantiuchina J, Lanza M, Bavota G (2018) Improving code: The (mis)perception of quality metrics. In: Proceedings of the 34th IEEE international conference on software maintenance and evolution, pp 80–91
Peters R, Zaidman A (2012) Evaluating the lifespan of code smells using software repository mining. In: Proceedings of the 16th european conference on software maintenance and reengineering, pp 411–416
Plösch R, Dautovic A, Saft M (2014) The value of software documentation quality. In: Proceedings of the 14th International conference on quality software, pp 333–342
Rong G, Zhang H, Shao D (2014) Investigating code reading techniques for novice inspectors: an industrial case study. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering, p 33
Sadeghzadeh Hemayati M, Rashidi H (2018) Software quality models: A comprehensive review and analysis. Journal of Electrical and Computer Engineering Innovations 6(1):59–76
Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: a case study at Google. In: Proceedings of the 40th international conference on software engineering: software engineering in practice, pp 181–190
Scalabrino S, Bavota G, Vendome C, Linares-Vasquez M, Poshyvanyk D, Oliveto R (2019) Automatically assessing code understandability. IEEE Trans Softw Eng 47(3):595–613
Sutter H, Alexandrescu A (2004) C++ coding standards: 101 rules, guidelines, and best practices. Pearson Education
Sweller J, Ayres P, Kalyuga S (2011) Cognitive load theory. Springer
Tahir A, Bennin KE, MacDonell SG, Marsland S (2018) Revisiting the size effect in software fault prediction models. In: Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement, p 23
Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). Trans Softw Eng 43(11):1063–1088
Vermeulen A, Ambler SW, Bumgardner G, Metz E, Misfeldt T, Thompson P, Shur J (2000) The Elements of Java (TM) Style. Cambridge University Press
Wintus T, Manshreck T, Wright H (2020) Software Engineering at Google. O'Reilly, Sebastopol, CA, USA

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Jürgen Börstler[1] · Kwabena E. Bennin[2] · Sara Hooshangi[3] · Johan Jeuring[4] · Hieke Keuning[4] · Carsten Kleiner[5] · Bonnie MacKellar[6] · Rodrigo Duran[7] · Harald Störrle[8] · Daniel Toll[9] · Jelle van Assema[10]

Kwabena E. Bennin
kwabena.bennin@wur.nl

Sara Hooshangi
shoosh@vt.edu

Johan Jeuring
J.T.Jeuring@uu.nl

Hieke Keuning
h.w.keuning@uu.nl

Carsten Kleiner
carsten.kleiner@hs-hannover.de

Bonnie MacKellar
mackellb@stjohns.edu

Rodrigo Duran
rodrigo.duran@aalto.fi

Harald Störrle
Harald.Stoerrle@qaware.de

Daniel Toll
daniel.toll@lnu.se

Jelle van Assema
J.vanAssema@uva.nl

1 Blekinge Institute of Technology, Karlskrona, Sweden

2 Wageningen University and Research, Wageningen, The Netherlands

3 Virginia Tech, Falls Church, VA, USA

4 Utrecht University, Utrecht, The Netherlands

5 University of Applied Sciences & Arts Hannover, Hannover, Germany

6 St John's University, Queens, NY, USA

7 Aalto University, Helsinki, Finland

8 QAware GmbH, Munich, Germany

9 Linnæus University, Kalmar, Sweden

10 University of Amsterdam, Amsterdam, The Netherlands