The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment

Jürgen Börstler, *Member, IEEE* and Barbara Paech, *Member, IEEE*

Abstract—Software readability and comprehension are important factors in software maintenance. There is a large body of research on software measurement, but the actual factors that make software easier to read or easier to comprehend are not well understood. In the present study, we investigate the role of method chains and code comments in software readability and comprehension. Our analysis comprises data from 104 students with varying programming experience. Readability and comprehension were measured by perceived readability, reading time and performance on a simple cloze test. Regarding perceived readability, our results show statistically significant differences between comment variants, but not between method chain variants. Regarding comprehension, there are no significant differences between method chain or comment variants. Student groups with low and high experience, respectively, show significant differences in perceived readability and performance on the cloze tests. Our results do not show any significant relationships between perceived readability and the other measures taken in the present study. Perceived readability might therefore be insufficient as the sole measure of software readability or comprehension. We also did not find any statistically significant relationships between size and perceived readability, reading time and comprehension.

Index Terms—Software readability, software comprehension, software measurement, comments, method chains, experiment

1 Introduction

Software readability and comprehension are major software cost factors. Software maintenance accounts for 66-90 percent of the total costs of software during its lifetime [1] and around half of those costs are spent on code comprehension [2], [3], [4]. Furthermore, more than 40 percent of the comprehension time is spent on plain code reading [5]. Readability is therefore a key cost-driver for software development and maintenance.

Chen and Huang [6] claim that inadequate documentation and lack of adherence to common guidelines or best practices are the most important problem factors for maintenance. Extensive documentation can significantly support software maintenance, but the extra effort needed to produce the necessary documents pays off only long-term and only for complex maintenance tasks [7]. In practice, documentation therefore rapidly deteriorates [8]. Writing self-documenting code, instead of documenting ill-structured code, is proposed as a partial solution to this problem [9], [10]. This emphasizes the importance of readable and comprehensible code, in particular in the context of Agile/Lean development practices where extraneous documentation might be considered as waste [11].

There is a large body of literature on general coding guidelines or practices to improve code readability and comprehension [12], [13], [14] as well as specific rules, heuristics and guidelines to obtain "good" or "better" (object-oriented) design or code, e.g., design patterns [15], [16], design heuristics [17], [18], code smells and refactoring [19], [20], [21]. The actual factors that make software easier to comprehend are, however, not well understood. Furthermore, the factors can also have complex interactions.

We distinguish people, project, cognitive and software factors, where people factors comprise properties of people and software factors comprise properties of software, cognitive factors are derived from cognitive theories and project factors describe elements of the project environment which can ease comprehension (see Fig. 1). In this classification readability is a software factor. Examples of interactions can be found between complexity, size and readability. Reducing the complexity of a program will likely also affect its size. More comments or more white-space might increase a programs readability and comprehensibility, but also make it longer. Longer programs are, however, less readable and more difficult to comprehend [22].

In the present study, we investigate the role of source code comments and method chains in software readability and comprehension. Method chaining has been advocated as a programming style that leads to more compact and more readable code [23], [24]. Careless use of method chaining can lead to violations of the Law of Demeter (LoD) [17] though, which can lead to more defects [25]. In coding guidelines source code comments are advocated as "absolutely vital to keeping ... code readable"[1], but also that focus should be on code that clearly communicates intent and functionality to reduce the need for comments [26].

- *J. Börstler is with the Department of Software Engineering, Blekinge Institute of Technology, Karlskrona 37179, Sweden. E-mail: jurgen.borstler@bth.se.*
- *B. Paech is with the Department of Computer Science, Heidelberg University, Heidelberg 69118, Germany. E-mail: paech@informatik.uni-heidelberg.de.*

Manuscript received 12 Dec. 2014; revised 31 Oct. 2015; accepted 28 Jan. 2016. Date of publication 10 Feb. 2016; date of current version 23 Sept. 2016. Recommended for acceptance by R. DeLine.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2527791

1. http://google-styleguide.googlecode.com/svn/trunk/cppguide.html#Comments, last visited 2014-09-12.

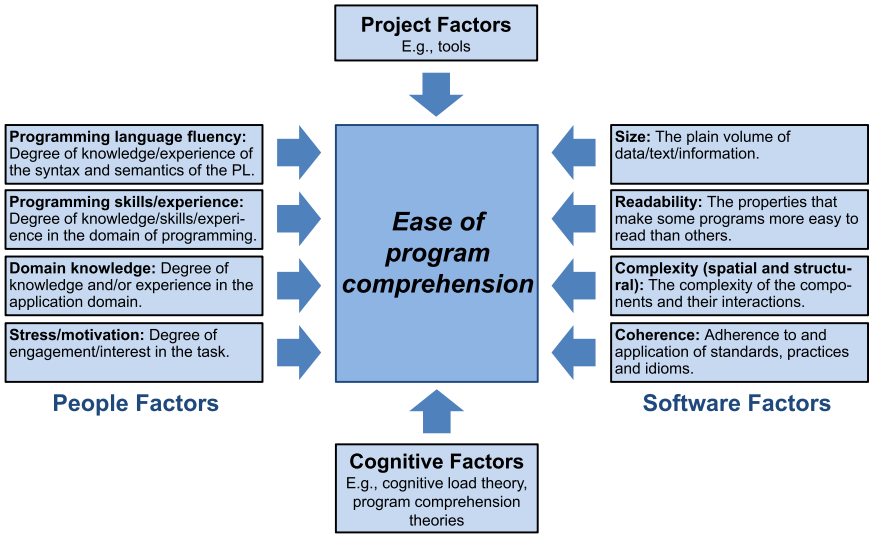


Fig. 1. Factors affecting the ease of program comprehension.

The remainder of the paper is roughly organized as proposed in common guidelines for empirical studies [27], [28], but has been slightly adapted for clarity of presentation. In the next section, we briefly review related work. In Section 3, we outline our research questions. The details of experiment planning and execution are described in Sections 4 and 5, respectively. Before a detailed analysis and discussion in Section 7, we give a brief overview over the raw data in Section 6. Threats to validity are discussed in Section 8. Lessons learned, conclusions and future work are presented in Sections 9 and 10, respectively.

2 Related Work

There is a large body of knowledge on methods, languages and tools to support program comprehension [29]. Although readability and comprehensibility are related, they are conceptually quite different. Readability is required for comprehensibility, but readability does not necessarily imply comprehensibility. That makes it difficult to measure readability objectively and independently of comprehensibility.

Smith and Taffler point out that in text readability studies comprehension is frequently used erroneously as a proxy for readability and that comprehension also is related to factors like context, education and experience [30]. In our work, we consider readability as a property of the code and comprehension as a characteristic of the reader. Klare points out though, that there is a strong relationship between text readability (as measured by readability formulas) and comprehension as well as reading speed [31]. Since reading speed can vary significantly between individuals, it needs to be calibrated carefully.

DuBay [32] defines readability as "what makes some texts easier to read than others. It is often confused with legibility, which concerns typeface and layout". Hargis [33] emphasizes that "[r]eadability depends on things that affect the readers' eyes and minds. Type size, type style, and leading affect the eye. Sentence structure and length, vocabulary, and organization affect the mind."

The focus of the present study is on the latter, inherent properties of the code, and we ignore legibility issues. Although, for example code coloring, can make code easier to read or understand, there are differences in readability and comprehensibility that cannot be alleviated by "things that affect the readers' eyes". While most editors have support for the handling of legibility issues like fonts and indentation, inherent code properties that affect readability and comprehension cannot be easily resolved using editors.

In the following subsections, we give a brief overview over the research that is related to the present study. Section 2.1 primarily focuses on recent studies on software readability. Sections 2.2 and 2.3 discuss related research on method chains and source code comments, respectively.

2.1 Software Readability and Comprehension

Readability has long been recognized as an important factor in software development [34], [35], [36]. A recent study at Microsoft showed that poor readability was ranked as the most important reason for initiating refactorings and improved readability the highest ranked benefit from refactoring [20].

There is only little research on measuring software readability [37], [38], [39]. Buse and Weimer proposed a measure for software readability based on the ratings of perceived readability of 120 students on 100 small code snippets in Java [38]. The code snippets were taken, as is, from five Open Source projects and are 4-11 lines in length, including comments. Indentation and white-space was not adjusted and snippets could comprise incomplete conditionals. A predictor was built using 25 features of those snippets, where the following features per line of code had the highest predictive power for readability (in decreasing order): average number of identifiers, average line length, average number of parentheses, maximum line length, and average number of '.'. The readability measure shows strong correlations with quality indicators like bugs indicated by FindBugs on 15 Open Source Java projects.

Posnett et al. [22] found several weaknesses in Buse and Weimer's model, most importantly that it does not scale well and that most of the variation could be explained by snippet size. They proposed a simpler readability model for Buse

and Weimer's dataset using 3 variables only; Halstead's Volume, lines of code, and token entropy.

Several studies have investigated identifier naming issues, e.g., [40], [41], [42]. We acknowledge that naming is an important factor for software readability and comprehension. In the present study, we focus on two additional important factors; comments and method chains. It should be noted that we do not aim at a general readability model like Posnett et al. or Buse and Weimer. A good overview over program comprehension models and early program comprehension experiments can be found in [43], [44].

2.2 Method Chains

Method chaining is an object-oriented programming style [23, Ch. 35]. A method that returns an object can be used as the source for another method call, as in the general example below.

```
object.method1(...).method2(...).method3();
```

Method chaining has been advocated as a good programming style [23], [24] and is used frequently to support more compact code as in the following examples.

```
/* (1) Method chain with identical method calls. */
StringBuffer sb = new StringBuffer(...);
sb.append(''Hello '').append(aNameString).append(''!'');

/* (2) All method calls return the same type. */
Scanner reader = new Scanner(...);
String inputLine = reader.nextLine().trim().toLowerCase();

/* (3) Unclear return types.                  */
/*    Might violate the Law of Demeter.    */
if (scanner.recordLineSeparator) {
   compilationUnit.compilationResult.lineSeparatorPositions
      = scanner.getLineEnds();
}
```

This can be intuitive when methods are chained in a systematic and predictable way, as in examples (1) and (2) above or so-called fluent interfaces [24]. If methods are chained ad hoc, as in example (3), method chaining might lead to less intuitive code and also to violations of the LoD [17]. In short, the LoD requires that a client object must only send messages to objects that are in its immediate scope, which enforces information hiding and makes all coupling explicit.

Guo et al. show that violations of the LoD lead to more defects [25][2]. Guo et al.'s study also shows that violations of the LoD are very common in the Eclipse plugins they evaluated. Marinescu and Marinescu show that clients of classes that exhibit design flaws are more fault-prone [45]. Thus, some forms of method chaining are more fault-prone and might be more difficult to understand.

2.3 Comments

Source code comments are highlighted in many coding guidelines as an important tool for program comprehension [46], [47]. There are, however, few empirical studies on the effects of source code comments on program comprehension. Furthermore, most of these studies are more than 20 years old.

Experimental studies from the 1980s show that the effect of source code comments on comprehension interacts with program decomposition and program indentation. Higher degrees of decomposition decreased the effects of commenting on program comprehension [48], [49]. Experiments by Norcio, revealed the best comprehension results for indented programs with single lines of comments interspersed with the code [50].

In a more recent experiment, Takang et al. showed that comments significantly improved program comprehension independently of the identifier naming style used (full vs. abbreviated names) [51]. This experiment also showed that full name identifiers were perceived as significantly more meaningful. There were no significant differences, though, in the comprehension of the programs with full and abbreviated names, respectively. The authors surmise that the programs used in the experiment might have been too familiar and the time given too long to give significant results in the test scores.

In another study, Nurvitadhi et al. investigated the utility of class and method comments in Java [52]. Compared to a program without any comments, method comments improved comprehension significantly, but class comments did not. Thus, as for method chains, some forms of comments might be more helpful for code comprehension than others.

3 Research Questions

In the present study we investigate in which ways commenting and method chaining affect software readability and comprehension.

RQ1: How does the amount and quality of source code comments affect software readability and comprehension?

RQ2: How does method chaining affect software readability and comprehension?

4 Experiment Planning

In the following, we describe the subjects, materials, tasks, dependent and independent variables, as well as the experiment design. We deliberately did not only measure code comprehension through readability scores by the subjects. We wanted to get an understanding of what the subjects have understood from the code. Therefore, we also used open questions where subjects had to summarize their code understanding as well as cloze questions where students had to recall the code to fill in gaps (see Sections 4.2.2 and 4.3.).

4.1 Subjects

The subjects were first and second year Computer Science students from Heidelberg University. The first year students participated in a course (with tutorials) covering a general introduction to programming and C++ in particular. The second year students participated in a course (with tutorials) covering a general introduction to software engineering which included a crash course in Java at the beginning. At the end of their courses, both groups got as a homework exercise to participate in the experiment and to reflect on the experiences with it. The students had to successfully complete 50 percent of all homework exercises. As

2. Example code (3) is a simplified version of a violation of LoD in the JDT core presented in [25].

TABLE 1
Key Characteristics of the Code Snippets Used in the Experiment (Variant with MCs and Good Comments)

Snippet	Source*	LOC	MC-un	CD	ExtCC	PHD	Description
S1	Web4J	22	1	0.405	1	2.19	Shortest method. One MC with four elements; get- and set-methods only. No conditionals (ExtCC=1). High PHD-readability.
S2	UniCase	54	1	0.694	7	4.81	Longest method. Most heavily commented. Nested conditionals (four levels). Three almost identical MCs with three elements each. Complex according to ExtCC, but highest PHD-readability.
S3	UniCase	46	4	0.338	7	−8.50	Long method. Nested conditionals (three levels) inside a loop. 11 partly similar method chains with three-four elements each; many get-methods, most often empty parameter lists, but 1 nested MC. Complex according to ExtCC and lowest PHD-readability.
S4	RaptorChess	36	4	0.341	4	−5.65	Medium size method. Three loops, no nesting. Four MCs with four elements each that all are comprised of `append`-calls, often with complex parameter lists. One nested MC. Average complexity. Low PHD-readability.
S5	Eclipse.jface	36	2	0.564	4	−5.87	Medium size method. One loop with nested conditional. Three largely similar MCs with three elements each; last MC-element is an attribute. Average complexity. Low PHD-readability.

**Fully qualified method names and links to the original source code can be found on the supplementary web page.*
LOC: Total lines of code, incl. empty lines and comments.
MC-un: Number of unique MCs.
CD: Comment density; comment character per non-comment character inside method body.
ExtCC: Extended cyclomatic complexity. ExtCC extends cyclomatic complexity by taking into account the complexity of the boolean expression in each branch.
PHD: Posnett et al.'s readability score as described in [22 Sect. 4.5] Higher scores indicate higher readability.

this was at the end of the course, only very few students really needed to complete this homework to reach this threshold. Thus, they were encouraged by specific emails to participate. Participation was therefore mainly voluntarily.

It should be noted that 42.3 percent of the students declared that they have high or very high practical experience from other languages than Java or C++. Furthermore, 19.2 percent of the students declared practical experience (medium-very high) as a professional programmer (see Fig. 13 in Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2016.2527791).

4.2 Materials

The following subsections describe the code snippets, comprehension questions, and other questions used in the present experiment. The full set of experiment materials can be downloaded from http://www.bth.se/com/jub.nsf. The key characteristics of the used code snippets are summarized in Table 1.

4.2.1 Code Snippets

The code snippets used in the experiment should be as realistic as possible, but still sufficiently general, representative and simple. The subjects should not need specific domain or application knowledge to understand them. To increase generalizability, we strived for code snippets that differ in their expression of comments and method chains, as well as in overall length and complexity. The code snippets should also vary in terms of existing readability measures (e.g., [22]). We therefore mined public Java projects for actual examples that we then adapted in the following way to suit the experiment context:

- Delete complex syntactical structures that are irrelevant for those parts of the code that are studied, e.g., inner classes or try/catch blocks.
- Replace unnecessarily cryptic identifiers by more intuitive and shorter ones. However, we tried to retain even lengthy names to avoid the breaking of naming patterns.
- Use camelCase-style for all identifiers.
- Remove all comments, except strategic comments[3].
- Introduce line breaks to keep line lengths below 80.
- Format all code according to the same style (K&R-style [53]).

Thus, we tried to minimize the influence of factors different from method chains and comments, such as naming style, line length or indentation. In contrast to the study of Buse and Weimer [38], we strived for self-contained code snippets (complete methods) with consistent formatting and indentation. We wanted to ensure that the snippets are readable as such to be able to isolate the influence of method chains and comments. Table 1 summarizes the key characteristics of code snippets S1-S5. An example of a code snippet and some of its variants can be found in Appendix A, available in the online supplemental material. As can be seen, the snippets vary in length and complexity as well as in the number of method chains and comments.

Each of the code snippets was then modified in a systematic way according to our experiment factors; method chains and comments. Regarding method chains, we developed following variants:

1) *MC (method chains)*. An original (adapted) method containing at least one method chain with three or more elements.
2) *NoMC (no method chains)*. A variant of the original as above, but with all method chains resolved. Method chains with more than two elements were broken up into several statements. If necessary, temporary

3. A strategic comment describes the purpose of a piece of code and is placed before this code.

variables were introduced. Existing variables were used where possible.

Regarding comments, we developed the following variants:

1) *GC (good comments)*. An original (adapted) method with useful strategic comments that give additional information beyond the actual code it explains.

```
/* Add all available analysis data (sublines). */
for (SublineNode subline : move.getSublines()) {...}
```

2) *BC (bad comments)*. A variant of the original as above, but with all source code comments replaced by comments that just repeat what the code does without explaining its purpose.

```
/* Add sublines. */
for (SublineNode subline : move.getSublines()) {...}
```

3) *NC (no comments)*. A variant of the original as above, but with all source code comments removed.

In all variants we retained the comments preceding the method header to convey the general purpose of the code of a snippet in the same way. Considering all combinations, we had 6 variants per snippet and thus 30 different snippets altogether. A comprehensive summary of measures and properties for all variants can be found in Table 6 in Appendix B, available in the online supplemental material.

4.2.2 Comprehension Questions

For each code snippet, we developed cloze tests to measure comprehension. In a cloze test certain parts of the text (code in our case) are blanked out and the subject has to fill in the blanks with suitable code, but not necessarily the original code. In contrast to free-form descriptions of the code content (which we also asked from the subjects), cloze tests allow a more standardized way of testing comprehension. If a subject has understood the overall purpose, behavior and flow of the code, it will be easier to provide an answer that is syntactically and semantically correct. This is, of course, easier than recalling the code or its structure verbatim. Such tests have long been used successfully in text comprehension tests and have also been shown applicable in tests of program comprehension [50], [54], [55].

In each code snippet, we blanked out the code that dealt with method chains (in the MC versions) and the code replacing the method chains (in the NoMC version), respectively. To make it difficult for the subjects to identify patterns in the blanked out parts of the code, we also blanked out unrelated code in some snippets. This resulted in two to six "gaps" for our snippets, depending on the complexity and number of method chains that were present in the particular snippet. An example of the gap placement for S1 is shown in Appendix A, available in the online supplemental material.

4.2.3 Background/Experience Questions

As recommended by Siegmund et al., we used self-estimation to judge subjects' overall programming experience and task-specific experience [56]. Furthermore, we asked subjects for their gender, whether they have a reading disorder, and for their identifier naming-style preference.

The actual questions used in the survey regarding task-specific experience and task-specific experience can be found in Fig. 14 in Appendix C, available in the online supplemental material.

4.3 Tasks

Subjects were shown a series of code snippets where each code snippet was shown twice. First, subjects were asked to carefully read through a snippet and assess its readability (reading task). Furthermore, subjects were asked to justify their assessment and summarize the main steps of the shown code (initial assessment). Second, we administered a simple cloze test (see Section 4.2.2). The subjects were shown the same snippet again, but with some parts left blank, which they had to fill in with the correct code (completion task). After the completion task they were (again) asked to assess the snippet's readability and to justify their assessment. They could also provide additional comments (follow-up assessment).

4.4 Dependent and Independent Variables

The independent variable of this experiment is the variant of the code snippet under investigation.

To capture code readability and comprehension, we measured the following dependent variables: Perceived readability on a scale of 1..5 (similar to [38]) after the reading task and after the completion task (R1 and R2, respectively); time in seconds to read the code and to complete the code (T_r and T_a, respectively); and accuracy of the completion task (Acc). According to Kintsch and Vipond, "reading time, recall and question answering are probably the most useful measures available" for readability and comprehension [57].

R1 captures the first impression of perceived readability for the subjects, while R2 captures the adjustment made to this impression based on the experience with the cloze-test. T_a and Acc indicate the "quality" (accuracy and speed) of the recall during the cloze-test, and thus the comprehension. We recorded T_r and T_a, respectively as the time taken from beginning a task to its end. However, we could not control whether the subjects actually spent their time on the tasks or not. For this reason, we excluded obvious outliers from the data.

Acc was measured in terms of how accurately the gaps of a snippet variant were completed. The researchers developed a scoring scheme for assessing correctness (0-3 points per gap) and assessed all gaps independently of each other. Conflicts were resolved by discussion. Acc was then defined as the ratio of scored points and total possible points, i.e. a number in the range [0..1].

The free-form answers from the initial and follow-up assessment (see Section 4.3) were not used in the present analysis.

Furthermore, we collected personal data from the subjects (gender, reading disorders), as well as data about their general programming experience, task related experience, and identifier naming-style preference. These data might affect how subjects perceive readability as well as their task performance.

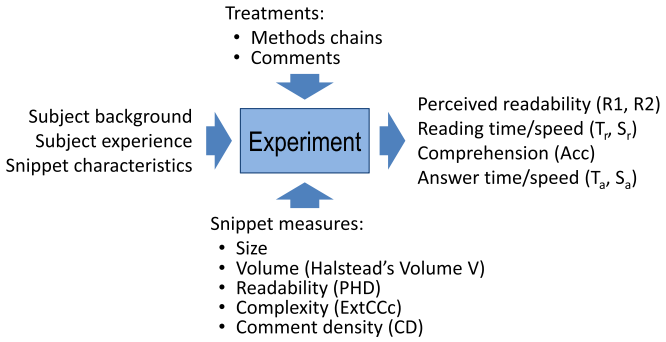


Fig. 2. Overview over all variables considered in the experiment.

An overview of all variables is shown in Fig. 2.

4.5 Design

Our experiment investigates two factors (method chains and source code comments), with two and three treatments, respectively.

Variants of code snippets were developed as outlined in Section 4.2.1.

We used a 2x3 factorial design with blocking. To mitigate ordering effects subjects were randomly assigned to one of six predefined snippet sequences. In each sequence, all snippets S1-S5 were shown in the same order, but in different variants. No variant was shown twice in any sequence. The sequences are shown in Table 2. The variants for method chains and comments are numbered from 1 to 2 and 1 to 3, respectively. Thus, Si_m:c describes the variant from snippet Si using method chain variant m and comment variant c (see also Section 4.2.1).

Subjects were randomly assigned to one of six predefined snippet sequences to ensure that all subjects see each snippet and each variant exactly once.

4.6 Piloting

Initially, we used six snippets with 36 variants in total. After a pilot study, we removed one snippet to cut down the expected total time for the experiment to at most 45 minutes, so that the experiment could be run within a traditional lecture.

5 Experiment Execution

The experiment was administered as an on-line questionnaire and instrumented using LimeService[4], see Fig. 3 for an overview. The questions regarding task-related experience and identifier naming-style preference were placed after the experimental tasks, since they might have influenced the subjects' answers.

LimeSurvey was also used for time-logging. For each subject we logged the time for each reading task and each completion task. The students were informed about the time-logging and that their answers and timing data "will only be used to study the readability and comprehensibility of code and not to assess your performance."

To mitigate fatigue effects and to make the time measurements more reliable, students were instructed to pause only at pre-defined breakpoints. They were also instructed that they cannot go back in the questionnaire and that they should "not take notes or copy the code snippets (manually or electronically), otherwise your answers would be useless for the study".

The instructions and full set of questions can be downloaded from http://www.bth.se/com/jub.nsf.

4. http://www.limeservice.com.

TABLE 2
Snippet Sequences Used in the Experiment

	Snippet				
Sequence	S1	S2	S3	S4	S5
Seq 1	S1_1:1	S2_2:2	S3_1:3	S4_1:2	S5_2:3
Seq 2	S1_1:2	S2_2:3	S3_1:1	S4_1:3	S5_2:1
Seq 3	S1_1:3	S2_2:1	S3_1:2	S4_1:1	S5_2:2
Seq 4	S1_2:1	S2_1:2	S3_2:3	S4_2:2	S5_1:3
Seq 5	S1_2:2	S2_1:3	S3_2:1	S4_2:3	S5_1:1
Seq 6	S1_2:3	S2_1:1	S3_2:2	S4_2:1	S5_1:2

Si_m:c – i:snippet no; m:1=MC,2=NoMC; c:1=GC,2=BC,3=NC. See Sect. 4.2.1 for an explanation of the acronyms.

6 Results

Overall, 255 subjects started the survey and 110 (43.1 percent) successfully completed it. Of those, we deleted six outliers, i.e. subjects with extremely short times for code reading and questions answering. The remaining 104 subjects provided 520 datapoints in total; 104 per snippet and between 14 and 23 for each individual snippet variant[5]. The median time for these 104 subjects for completing the survey was 48.5 minutes (including pauses). For the present analysis, we only included the data from those 104 subjects.

Fig. 4 gives an overview over the subjects' demographics. The data shows that the majority of subjects is male (83.7 percent) and that the majority of subjects have a preference for camelCase-style format (70.2 percent). In general, subjects have a high overall programming experience (43.3 percent), but a low overall task-specific experience (53.8 percent).[6] Only three subjects (2.9 percent) declared a reading disorder. Since their data were no outliers, we included them in the analysis.

Table 3 summarizes the data for perceived readability (R1, R2), timing data (T_r, T_a, S_r, S_a) and answer accuracy (Acc) for all 30 code snippet variants. The raw data for all completed answers can be downloaded from http://www.bth.se/com/jub.nsf.

7 Analysis and Discussion

In the following, we first describe some preliminaries on how we analyzed the results. Then we discuss the major results with respect to the overall influence of method chains and code comments, of subject characteristics and of the snippets on perceived readability and comprehension. We also discuss the relationships between different experiment variables

5. The imbalance of datapoints per snippet variant is due to an over-representation of a specific snippet series among the excluded subjects.

6. Overall programming experience is aggregated from the subjects' responses regarding experience levels in Java, C++, and Other programming languages. Overall task-specific experience is aggregated from the subjects' responses regarding knowledge/experience in OOD, LoD, refactoring and plug-in programming in Eclipse, see Appendix C, available in the online supplemental material, for details.

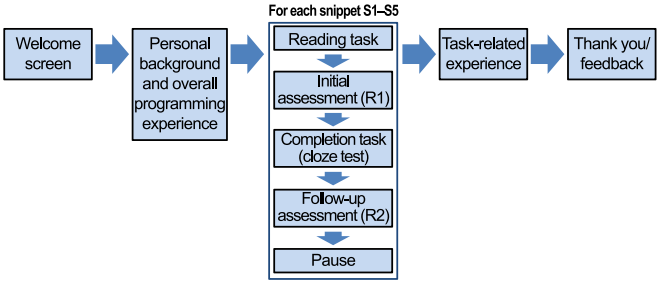


Fig. 3. Overview over the on-line questionnaire. Timing data was taken for each of the "boxes".

and go into detail on different subject groups and snippet variants in subsections. A summary of all relationships found in the present experiment is shown in Fig. 5.

Perceived readability (R1, R2) was measured on a scale from very difficult to very easy, i.e. these data are ordinal. Subjects were asked to rate snippet readability based on their own programming experience. Absolute individual scores are therefore less relevant than relative differences. Differences in perceived readability between groups of subjects or snippets/snippet variants were tested using Chi-Square tests (χ^2).

Stacked bar charts as in Figs. 6 and 7 are used to visualize the distribution of actual scores of R1. Each bar represents a total (100 percent) and each part shows the proportion of scores in a category. Each bar is centered at 0 percent which makes it easier to compare the relative perceived readability of a total.

Since R1 and R2 are strongly and significantly related according to Spearman's rank correlation ($\rho = 0.848; \alpha < 0.0001$), we ignore R2 in our further analysis

Method chains and comments. Regarding RQ2, our data does not show any significant differences in the perceived readability (R1) for the MC variants. Regarding RQ1, there are significant differences between the comment variants ($\chi^2 = 16.1$; $\alpha = 0.003$). Code snippets with good comments (GC) are perceived as the most readable and the variants without comments (NC) are perceived as the least readable. The Acc means for the MC and comment variants are all between 0.43 and 0.45. All differences are insignificant (see Fig. 6).

Subject characteristics. When looking at different subject groups (see Fig. 7), we can identify differences in perceived readability for several subject groups. For example, there is a significant relationship between overall programming experience and R1 ($\chi^2 = 19.7; \alpha = 0.001$) as well as between task-specific experience and R1 ($\chi^2 = 29.7; \alpha < 0.0001$). ANOVA tests show that also the means for Acc are significantly higher for the groups with high overall programming and high task-specific experience ($\alpha < 0.01$).

Regarding naming preferences, our data shows a significant difference in R1 between the subject group that has a naming preference and the groups that have none ($\chi^2 = 9.55; \alpha = 0.008$). The difference between the two preference groups (camelCase-style versus under_score-style) is also significant ($\chi^2 = 12.7; \alpha = 0.013$). Overall, the student group without a naming preference finds the snippet variants more difficult to read than the other

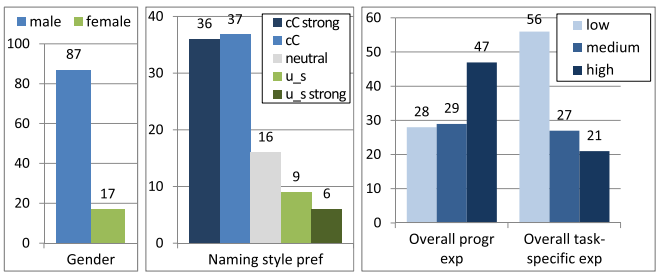


Fig. 4. Gender, naming preference (cC=camelCase style, u_s=under_score style) and overall experience levels (programming and task-specific) for the subjects (in absolute numbers for all 104 subjects).

TABLE 3
Perceived Readability (R1, R2), Timing Data (T_r, T_a, S_r, S_a) and Answer Accuracy (Acc) for All Snippets

Snippet	N	R1	R2	T_r	T_a	Acc	S_r	S_a
S1_1:1	16	2.81	2.38	79.11	135.25	0.40	8.30	4.86
S1_1:2	23	2.74	2.70	70.73	101.92	0.38	8.75	6.07
S1_1:3	14	2.71	2.57	82.21	81.02	0.38	6.35	6.44
S1_2:1	15	2.60	2.27	88.14	92.13	0.27	8.10	7.75
S1_2:2	21	3.00	2.71	101.62	108.17	0.41	6.65	6.25
S1_2:3	15	2.47	2.20	84.41	104.28	0.33	6.86	5.55
S2_1:1	15	2.60	2.53	183.55	113.67	0.50	7.77	12.55
S2_1:2	15	2.60	2.47	153.24	89.51	0.61	8.14	13.93
S2_1:3	21	2.29	2.24	113.28	80.97	0.65	8.08	11.30
S2_2:1	14	2.43	2.57	187.26	140.11	0.52	7.99	10.68
S2_2:2	16	2.00	1.94	120.97	206.93	0.51	10.89	6.36
S2_2:3	23	2.35	2.26	153.01	104.24	0.53	6.44	9.45
S3_1:1	23	3.35	3.00	136.66	87.09	0.48	8.56	13.43
S3_1:2	14	2.64	2.29	124.64	104.80	0.41	9.47	11.26
S3_1:3	16	2.25	2.25	153.84	91.50	0.46	5.86	9.85
S3_2:1	21	3.00	2.86	179.61	106.91	0.40	6.32	10.62
S3_2:2	15	2.33	1.87	79.05	141.04	0.37	14.48	8.12
S3_2:3	15	2.73	2.33	54.63	85.83	0.28	15.85	10.09
S4_1:1	14	2.79	2.29	143.10	67.07	0.37	6.32	13.48
S4_1:2	16	2.94	2.69	119.32	60.22	0.49	6.65	13.17
S4_1:3	23	2.30	2.09	98.62	55.68	0.43	7.31	12.95
S4_2:1	15	2.93	2.87	145.29	84.36	0.37	6.86	11.81
S4_2:2	15	3.00	2.80	72.49	50.74	0.44	12.21	17.44
S4_2:3	21	2.67	2.48	140.08	62.29	0.52	5.80	13.05
S5_1:1	21	3.38	3.38	114.36	71.54	0.47	8.00	12.79
S5_1:2	15	2.40	2.60	113.77	73.08	0.37	7.62	11.86
S5_1:3	15	2.93	3.00	74.18	45.27	0.28	9.32	15.26
S5_2:1	23	3.13	3.00	130.16	75.75	0.53	7.44	12.79
S5_2:2	14	2.93	2.57	118.97	92.26	0.39	7.74	9.98
S5_2:3	16	2.75	2.56	116.50	79.42	0.46	6.40	9.38
ALL	520	2.72	2.54	108.32	88.76	0.44	8.22	10.62

N: Number of datapoints (subjects).
R1,R2: Average perceived readability after the reading and completion task.
T_r, T_a: Median snippet reading and answering time in seconds.
Acc: Average answer accuracy in percent.
S_r, S_a: Median reading and answering speed in characters per second.

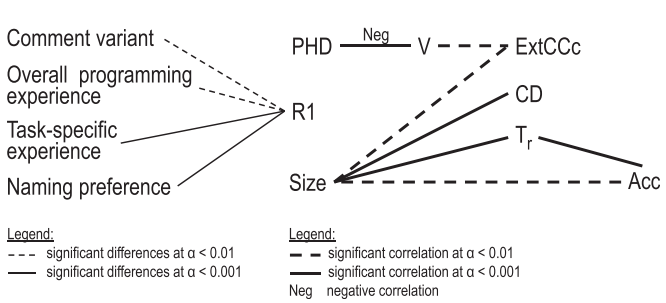


Fig. 5. Summary of relationships between experiment variables.

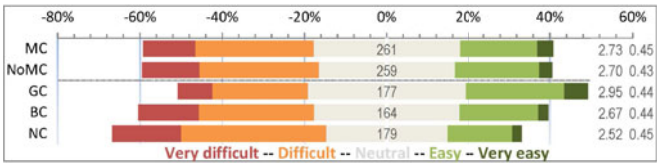


Fig. 6. Distribution of scores for perceived readability (R1) for method chain variants (MC/NoMC) and comment variants (GC/BC/NC). The numbers in the middle show the number of datapoints for each variant. The numbers in the two columns to the right show the average perceived readability (R1, left column) and the average answer accuracy (Acc, rightmost column).

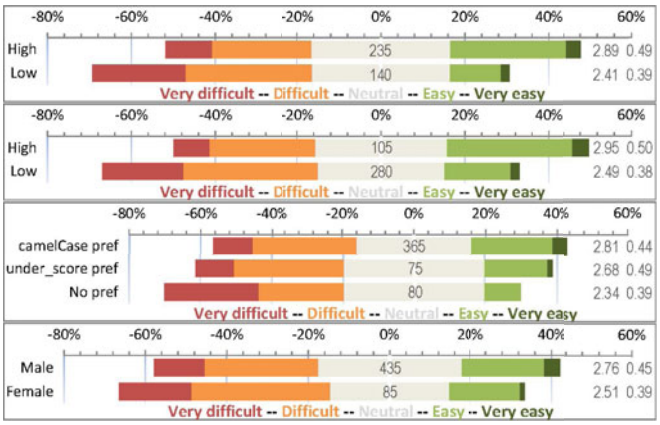


Fig. 7. Distribution of scores for perceived readability (R1) for different subject groups (from top to bottom): Overall programming experience, overall task-specific experience, identifier naming preference and gender.

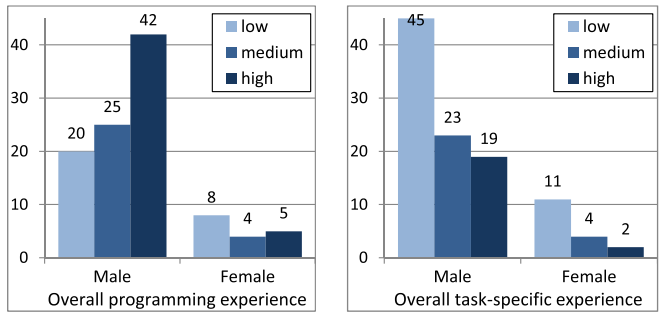


Fig. 8. Self-assigned experience levels for male and female subjects (in absolute numbers for all 104 subjects).

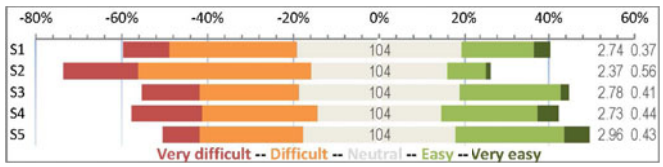


Fig. 9. Distribution of scores for perceived readability (R1) for snippet S1-S5.

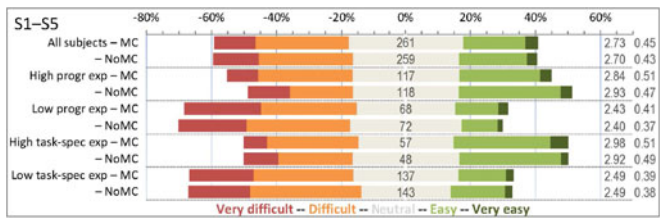


Fig. 10. Distribution of scores for perceived readability (R1) for all snippets by MC variant, grouped by subject group experience level.

groups and also has the lowest answer accuracy (Acc). The differences between groups with respect to Acc are, however, insignificant.

Fig. 7 shows that, overall, male subjects give higher readability scores than females and have a higher answer accuracy. Both difference are not statistically significant, though.

The self-assigned experience levels for men and women do not differ much, except for experience in programming languages other than Java and C++ ("Other lang exp" in Fig. 13, in Appendix C, available in the online supplemental material). The aggregated overall experience levels are slightly lower for women than for men. None of these differences in experience (see Fig. 8) are statistically significant though, according to a Fisher's exact test.

Code snippets. For snippets S1-S5, our data shows significant differences in the overall perceived readability (R1) ($\chi^2 = 22.8; \alpha = 0.004$) as well as in their mean answer accuracy (Acc) (ANOVA $p = 0.0013$ at $\alpha < 0.01$) (see Fig. 9). That is, our snippets were sufficiently different to lead to significant differences in the independent variables.

Table 7 in Appendix D, available in the online supplemental material, shows the Spearman rank correlations (ρ) for the scores and timing data (Table 3) and measurements for all snippet variants (Table 6 in Appendix B, available in the online supplemental material). It does not shows any significant relationships at the $\alpha < 0.01$-level between R1 and timing data (T_r, T_a, S_r, S_a) or Acc. That is, perceived readability does not correlate with traditional measures of text readability or comprehension. Studies on software readability might therefore be improved by using measures in addition to perceived readability (see also the discussion on bias in Section 8.1).

R1 also does not show any significant relationships with size, volume (Halstead's Volume V), complexity (ExtCCc), or comment density (CD). Regarding Posnet et al.'s readability measure (PHD), our dataset does not show any significant relationship between PHD and any other measure, except V ($\rho = 0.898; \alpha < 0.001$). One should note though, that V is a factor in the formula to compute the PHD measure.

For our dataset, neither perceived readability (R1) nor PHD are good predictors for other measures of readability or comprehension. In particular, there is no significant relationship between size and R1, as for example for the Buse and Weimer dataset (as shown in [22]).

However, our data shows that snippet size correlates strongly and highly significantly with reading time (T_r) ($\rho = 0.689; \alpha < 0.001$) and moderately and significantly with Acc ($\rho = 0.512; \alpha < 0.01$). Furthermore, there is a moderate and highly significant positive relation between T_r and Acc ($\rho = 0.555; \alpha < 0.001$). That is, for our dataset, we can see that larger snippets tend to have longer reading times, but also higher answer accuracies. Neither of those have a significant relationship with perceived readability, though. Since our subjects had unlimited time for reading and answering, potentially negative impacts of size on readability and comprehension could be compensated by spending more time on larger code snippets. This might have affected their performance on the cloze test. On the other hand, our

TABLE 4
Differences in R1 and Acc between Subject Groups with Low and High Experience for the Snippets' MC Variants

Snippet	All subjects	High exp groups*	Low exp groups*	Comment
S1	Similar R1 for MC and NoMC. Slightly higher Acc for MC.	Inconsistent, but less readable variant has higher Acc.	NoMC more readable and slightly higher Acc.	Most groups consider NoMC more readable. Slightly higher Acc for NoMC for most groups.
S2	MC more readable and higher Acc.	MC more readable. Inconsistent for Acc.	MC more readable and higher Acc.	All groups consider MC more readable. Higher Acc for MC for all but the smallest group (high task-spec exp).
S3	MC more readable and higher Acc.	Inconsistent.	MC more readable and higher Acc.	All but one group consider MC more readable. Higher or same Acc for MC for all groups.
S4	NoMC more readable and slightly higher Acc.	NoMC more readable, but lower Acc.	NoMC more readable and higher Acc.	All groups consider NoMC more readable. High and low exp groups contradictory regarding Acc, but overall slightly higher Acc for NoMC.
S5	Similar R1 for MC and NoMC. Higher Acc for NoMC.	NoMC more readable and higher Acc.	Inconsistent	All groups, except the low task-specific exp group consider NoMC more readable. All but the low progr exp group have higher Acc for NoMC.

**There are two such groups: High/low programming experience and high/low task-specific experience, respectively.*

data shows negative correlations between times and speeds for reading $(T_r, S_r; \rho = -0.491, \alpha < 0.05)$ and answering $(T_a, S_a; \rho = -0.694, \alpha < 0.001)$, respectively. I.e. on larger snippets, the subjects were still faster in terms of snippet characters per second.

Taken together, we can conclude that there are statistically significant differences in the perceived readability of the tested code snippets with respect to different comment variants (RQ1). There are no differences in answer accuracy for method chain or comment variants (RQ2). However, there are statistically significant differences between subject groups with low and high experience, respectively.

In the following subsections, we look at the different subject groups and snippet variants in some more detail.

7.1 Method Chains: All Snippets

As already shown in Fig. 7, there is a notable difference between the subject groups with high and low programming and task-specific experience, respectively. Subjects with high experience rate the code snippets, overall, as more readable than subjects with low experience and have higher Acc-values. Within an experience group the differences in R1 and Acc are marginal. An ANCOVA analysis for Acc with overall general and task-specific experience levels[7], respectively, as covariates shows that the observed means for Acc are almost identical to the adjusted means.

7.2 Method Chains: Individual Snippets

When we break down Fig. 10 to the level of individual snippets, we get larger differences for R1 and Acc within subject groups. These differences do not follow a consistent pattern for all snippets, though. Furthermore, none of the differences within an experience group is statistically significant. This observation also holds when looking at the method chain variants independently of the comment variants.

A summary of the observations from breaking down the analysis to snippet level and experience groups can be found in Table 4. As already indicated in Fig. 9, there are considerable differences between the snippets. For snippets S2 and S3, the experiment results show an advantage for the MC variants for R1 as well as for Acc. For snippets S1, S4 and S5, the results are almost the opposite. In large, the snippet variants with higher R1 also have higher Acc.

From the available data it is not clear whether these differences are related to specific properties of the actual method chains in S2 and S3 on the one hand and in S1, S4 and S5 on the other. We can note though, that S3 is the snippet with the most method chains and the only snippet where the NoMC variant is smaller than the MC variant.

The role of such properties should be studied in more detail.

7.3 Comments: All Snippets

For the comment variants, as for the MC variants, there are notable difference between the groups with high and low programming and task-specific experience, respectively (see Fig. 11). Contrary to the MC variants, we can see considerable differences within different experience groups. Except for the low programming experience group, all

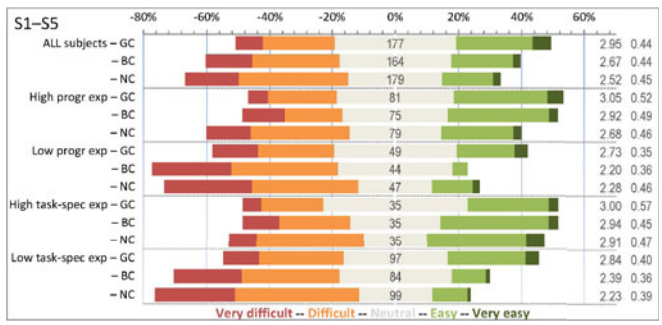


Fig. 11. Distribution of scores for perceived readability (R1) for all snippets by comments variant grouped by subject group experience level.

7. For the ANCOVA analysis, we used the weighted sums of the first six experience indicators in Fig. 13 in Appendix C, available in the online supplemental material, for general experience and the remaining four for task-specific experience.

TABLE 5
Differences in R1 and Acc between Subject Groups with Low and High Experience for the Snippets' Comment Variants

Snippet	All subjects	High exp groups*	Low exp groups*	Comment
S1	BC easiest to read; NC most difficult. Acc highest for BC; almost the same for GC and NC	BC easiest to read; NC most difficult. Inconsistent regarding Acc.	BC easiest to read; GC most difficult with many "very difficult" scores. Acc follows same pattern, i.e. BC>NC>GC.	BC has highest R1 and highest Acc in all groups. Inconsistent results for R1 and Acc regarding GC and NC.
S2	GC easiest to read; Small differences between BC and NC. Small differences in Acc (NC>BC>GC).	Inconsistent regarding R1. Lowest Acc for BC.	BC most difficult to read with very many "very difficult" scores. GC slightly higher Acc as NC.	BC most difficult to read in 4 of the 5 groups. Inconsistent regarding Acc but overall high.
S3	GC much easier to read than BC and NC, which are about the same. Similar for Acc, but differences are smaller.	GC easiest to read, many "very easy" scores; BC most difficult. Inconsistent regarding Acc but overall high.	GC much easier to read than BC and NC; NC most difficult with many "very difficult" scores. Lowest Acc for BC; Acc on an overall low level.	GC consistently rated as easiest to read; R1 inconsistent for BC and NC, but NC has the most"very difficult" scores in all groups. Inconsistent and large variations in Acc.
S4	BC easiest to read; NC most difficult. Acc almost the same for GC, BC and NC.	BC easiest to read, despite many "very difficult" scores. Inconsistent regarding DC and NC. Acc highest for NC and lowest for BC.	Inconsistent regarding R1, but differences are small. Very low Acc for GC; highest Acc for NC.	Large inconsistencies regarding R1 and Acc, but consistently highest Acc for NC.
S5	GC easiest to read; BC most difficult. Highest Acc for GC; almost the same for BC and NC.	GC easiest to read; inconsistent regarding BC and NC. Highest Acc for GC; lowest for NC. Comparatively small differences in R1, but large differences in Acc.	GC much easier to read than BC and NC; BC most difficult. Highest Acc for GC.	GC consistently rated easiest to read; BC most difficult in 4 of the 5 groups. Acc consistently highest for GC in all groups; lowest Acc for NC in 4 of the 5 groups.

**There are two such groups: High/low programming experience and high/low task-specific experience, respectively.*

groups perceive GC as most readable and NC as least readable. This challenges some earlier work on comments discussed in Section 2.3 and suggests that the role and quality of comments should be investigated in more detail.

For the low task-specific experience group, the differences are significant ($\chi^2 = 19.4; \alpha = 0.0007$). The differences in Acc between experience groups as well as within experience groups are small.

An ANCOVA analysis (as for the method chains in Section 7.1) shows no significant differences in the Acc means for the different comment variants.

7.4 Comments: Individual Snippets

When breaking down Fig. 11 to the level of individual snippets, we get a more inconsistent picture. Table 5 on the next page provides a summary of the observations for individual snippets. Overall, we can see that either GC or BC were rated as the most readable snippet variants. Regarding the Acc-values all variants were rated best or worst for some snippets. Unlike for the MCs, there does not exist a tendency that variants with a high R1 also have a high Acc. This raises the question whether the subjects actually considered the quality of the comments or just their presence or amount when rating the readability of the code.

8 Threats to Validity

8.1 Internal Validity

Internal validity is concerned with the observed relationships between the independent and dependent variables, i.e. to which extent the treatment and independent variable actually caused the observed effects. Unknown factors (confounding variables, bias, etc.) might have affected the results and limit (internal) validity of the study. As discussed in Section 4.2.2, we tried to minimize learning effects concerned with the cloze questions by varying the places of the gaps. We also reduced the number of snippets to 6 after piloting to bring down the expected total task time to under 45 minutes. Furthermore, we minimized fatigue effects by allowing pausing. The snippets were always shown in the same order (S1-S5), so that the results for the last snippets could be affected by fatigue. However, we placed the largest and most complex snippets in positions 2 and 3, respectively to mitigate this problem.

As shown in the discussion of the results, programming experience is a confounding factor for comprehension which we controlled for explicitly. We could not control cheating by our students. As the students participated voluntarily we see no reason for validity issues here.

A problem, that might have affected our results is bias. Since our subjects are undergraduate students, their perception of what is readable or not might be biased by personal beliefs or beliefs imposed on them (e.g., in their programming education). This should, however, only affect our measures of perceived readability. Since perceived readability and other measures of readability and comprehension do not show any significant relationships in our study, this threat might be real. Therefore perceived readability might not be sufficient as the sole measure of readability (see also the discussion in Section 7).

8.2 External Validity

External validity is concerned with the generalizability of the results to other contexts. Our study is restricted by the

fact that the subjects were undergraduate students and only few rated themselves with high experience. Results might be different for professionals. The snippets used in the study do not contain complex programming language features. This limits generalizability to code with common data structures and control logic. However, we believe that our snippets are representative for professional code with these constraints as we took them from known code bases and only made slight adaptations. The limited number of base snippets could also be seen as a threat to generalizability. However, the goal of the study was not to develop a general readability formula. We investigated the effect of specific code variations only. The code variations were carefully developed (see Section 4.2.1) and, as shown by the data, the snippets differ in their readability. Limiting the number of snippets allowed us to gather enough data points for the variants for statistical analysis.

As mentioned in Section 7, unlimited time might be an issue. In our experiment, reading and answer time were not restricted. For professional programmers, time on task is usually restricted. However, this allowed us to study the influence of the reading and answering times on the accuracy of the results.

8.3 Construct Validity

Construct validity is concerned with the operationalization of the study and to which extent the factors, measures, and materials actually represent the intended real world constructs. As the good and bad comments were partly defined by the researchers, it could be that they do not represent realistic code comments. However, we preserved existing comments as far as possible and used common commenting guidelines. We believe that the differences as described in Section 4 were quite typical.

Regarding measures, we deliberately used measures other than perceived software readability to also include other factors of comprehension. As the researchers devised the gaps for the cloze questions and rated the accuracy, it could be that by chance the gaps were too artificial. However, as mentioned in Section 4, we placed the gaps at very different places. Furthermore, we evaluated accuracy individually, based on assessment criteria defined and agreed upon in advance. This process led to a high agreement and only few answers required a discussion.

8.4 Conclusion Validity

Conclusion validity is concerned with the correctness of the conclusions drawn in this study. We used standard statistical algorithms as recommended in text books on experimentation in software engineering such as [28]. The online questionnaire made sure that the questions were administered uniformly. Ordering and learning effects were countered by grouping.

9 Lessons Learned

We want to group the lessons learned from this study into two groups: Lessons learned from the actual experiment process and lessons learned from the actual results. Both gave valuable insight into considerations for future experiments in software readability and comprehension.

- It is very difficult to isolate and study a single comprehension factor, since there are so many factors that interact with each other in ways that are not well understood. The experimental code must show as much variation as possible to be able to generalize, but as little as possible variation to be able to get statistically significant results.
- Timing is important. Giving subjects unlimited time for the experimental tasks makes measuring of comprehension very difficult. This cannot be made up for easily by taking into account the time on task.
- Perceived readability (as R1) might be insufficient as the only measure of readability and/or comprehension, since it can be affected by bias that is difficult to control. In the present study, it seems that the subjects were biased towards the existence of comments.
- Controlling for experience is very important even for single cohorts of students, since experience is a significant factor for comprehension.

10 Conclusions and Future Work

In the present study, we have shown that code comments on statement level affect the perceived readability of software, but not comprehension measured in terms of accuracy of answers to cloze questions. Regarding the former, our study shows that code with bad comments was rated as more readable than expected. In several cases the variants with bad comments were actually rated as the most readable variants. This suggests that the role of comments for code quality should be studied in more detail.

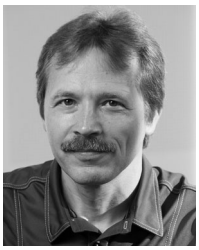
In our study, the absence or presence of method chains is not significantly related to perceived readability or comprehension. Method chaining is often claimed to lead to more compact and more readable code. This could not be corroborated by our study. In fact, it is interesting to note that even experienced subjects favored code without method chains in several cases (see Table 4). The ambiguous results of our study could be an artefact of differences in the code snippets used in the experiment. Further studies are necessary to provide empirical evidence for or against method chaining as an object-oriented programming style.

The result that perceived readability and comprehension (measured by answer accuracy—Acc) are not related is somewhat disturbing. It shows that it is difficult to measure readability and comprehension and to investigate their relationship. It could also mean that perceived readability might not be sufficient as a sole indicator or predictor of software quality. This should be studied in more detail.

References

[1] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, 2000.
[2] S. W. Yip and T. Lam, "A software maintenance survey," in *Proc. 1st Asia-Pacific Softw. Eng. Conf.*, 1994, pp. 70–79.
[3] J. R. Foster, "Cost factors in software maintenance," Ph.D. dissertation, School Eng. Comput. Sci., Univ. Durham, Durham, U.K., 1993.
[4] V. Nguyen, "Improved size and effort estimation models for software maintenance," in *Proc. 26th Int. Conf. Softw. Maintenance*, 2010, pp. 1–2.
[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 492–501.

[6] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *J. Syst. Softw.*, vol. 82, no. 6, pp. 981–992, 2009.
[7] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "The impact of UML documentation on software maintenance: An experimental evaluation," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 365–381, Jun. 2006.
[8] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Softw.*, vol. 20, no. 6, pp. 35–39, Nov./Dec. 2003.
[9] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*. New York, NY, USA: McGraw-Hill, 1978.
[10] D. Spinellis, "Code documentation," *IEEE Softw.*, vol. 27, no. 4, pp. 18–19, Jul./Aug. 2010.
[11] K. Petersen, "Implementing lean and agile software development in industry," Ph.D. dissertation, School Comput., Blekinge Inst. Technol., Karlskrona, Sweden, 2010.
[12] P. W. Oman and C. R. Cook, "A programming style taxonomy," *J. Syst. Softw.*, vol. 15, no. 3, pp. 287–301, 1991.
[13] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. New York, NY, USA: Pearson Education, 2004.
[14] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, and P. Thompson, *The Elements of Java (TM) Style*. Cambridge, U.K.: Cambridge Univ. Press, 2000.
[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY, USA: Wiley, 1996.
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1995.
[17] K. J. Lieberherr and I. M. Holland, "Assuring good style for object-oriented programs," *IEEE Softw.*, vol. 6, no. 5, pp. 38–48, Sep. 1989.
[18] A. J. Riel, *Object-Oriented Design Heuristics*. Reading, MA, USA: Addison-Wesley, 1996.
[19] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.
[20] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, Jul. 2014.
[21] J. Kerievsky, *Refactoring to Patterns*. Reading, MA, USA: Addison-Wesley, 2005.
[22] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proc. 8th Working Conf. Mining Softw. Repositories*, 2011, pp. 73–82.
[23] M. Fowler, *Domain-Specific Languages*. Reading, MA, USA: Addison-Wesley, 2010.
[24] Y. E. Keskin. (2014, Mar.). Fluent interface for more readable code. [Online]. Available: http://java.dzone.com/articles/fluent-interface-more-readable-0
[25] Y. Guo, M. Würsch, E. Giger, and H. C. Gall, "An empirical validation of the benefits of adhering to the law of Demter," in *Proc. 18th Working Conf. Reverse Eng.*, 2011, pp. 239–243.
[26] R. Green and H. Ledgard, "Coding guidelines: Finding the art in the science," *Commun. ACM*, vol. 54, no. 12, pp. 57–63, 2011.
[27] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering*. New York, NY, USA: Springer, 2008, pp. 201–228.
[28] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. New York, NY, USA: Springer, 2012.
[29] M.-A. Storey, "Theories, tools and research methods in program comprehension: Past, present and future," *Softw. Quality J.*, vol. 14, no. 3, pp. 187–208, 2006.
[30] M. Smith and R. Taffler, "Readability and understandability: Different measures of the textual complexity of accounting narrative," *Accounting, Auditing Accountability J.*, vol. 5, no. 4, pp. 84–98, 1992.
[31] G. R. Klare, "Readable computer documentation," *J. Comput. Documentation*, vol. 24, no. 3, pp. 148–168, 2000.
[32] W. H. DuBay, *The Principles of Readability*. Costa Mesa, CA, USA: Impact Inf., 2004.
[33] G. Hargis, "Readability and computer documentation," *J. Comput. Documentation*, vol. 24, no. 3, pp. 122–131, 2000.
[34] D. E. Knuth, "Literate programming," *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984.
[35] J. L. Elshoff and M. Marcotty, "Improving computer program readability to aid modification," *Commun. ACM*, vol. 25, no. 8, pp. 512–521, 1982.
[36] L. E. Deimel and J. F. Naveda, "Reading computer programs: Instructor's guide and exercises," Softw. Eng. Inst., Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-90-EM-3, 1990.
[37] J. Börstler, M. E. Caspersen, and M. Nordström, "Beauty and the beast—Toward a measurement framework for example program quality," Dept. Comput. Sci., Umeå Univ., Umeå, Sweden, Tech. Rep. UMINF-07.23, 2007.
[38] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 546–558, Jul./Aug. 2010.
[39] J. Börstler, M. E. Caspersen, and M. Nordström. (2015). Beauty and the beast: On the readability of object-oriented example programs, *Softw. Quality J.*. [Online]. Available: http://link.springer.com/article/10.1007/s11219-015-9267-5
[40] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: An analysis of trends," *Empirical Softw. Eng.*, vol. 12, no. 4, pp. 359–388, 2007.
[41] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Softw. Eng.*, vol. 18, no. 2, pp. 219–276, 2013.
[42] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, 2010, pp. 156–165.
[43] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *IEEE Comput.*, vol. 28, no. 8, pp. 44–55, Aug. 1995.
[44] F. Détienne, *Software Design–Cognitive Aspects*. New York, NY, USA: Springer, 2002.
[45] R. Marinescu and C. Marinescu, "Are the clients of flawed classes (also) defect prone?" in *Proc. 11th IEEE Int. Working Conf. Source Code Anal. Manipulation*, 2011, pp. 65–74.
[46] S. W. Ambler, A. Vermeulen, and G. Bumgardner, *The Elements of Java Style*. New York, NY, USA: Cambridge Univ. Press, 1999.
[47] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Boston, MA, USA: Prentice-Hall, 2008.
[48] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 215–223.
[49] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sep. 1988.
[50] A. Norcio, "Indentation, documentation and programmer comprehension," in *Proc. Conf. Human Factors Comput. Syst.*, 1982, pp. 118–120.
[51] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: An experimental investigation," *J. Program. Lang.*, vol. 4, no. 3, pp. 143–167, 1996.
[52] E. Nurvitadhi, W. W. Leung, and C. Cook, "Do class comments aid Java program understanding?" in *Proc. 33rd Annu. Frontiers Educ.*, vol. 1, 2003, p. T3C-13.
[53] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, vol. 2. Englewood Cliffs, NJ, USA: Prentice-Hall, 1988.
[54] C. Cook, W. Bregar, and D. Foote, "A preliminary investigation of the use of the cloze procedure as a measure of program understanding," *Inf. Process. Manage.*, vol. 20, no. 1, pp. 199–208, 1984.
[55] W. E. Hall and S. H. Zweben, "The cloze procedure and software comprehensibility measurement," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 5, pp. 608–623, May 1986.
[56] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Softw. Eng.*, vol. 19, pp. 1299–1334, 2014.
[57] W. Kintsch and D. Vipond, "Reading comprehension and readability in educational practice and psychological theory," in *Perspectives on Memory Research*, L.-G. Nilss, Ed., Mahwah NJ, USA: Lawrence Erlbaum Assoc., 1979.



Jürgen Börstler received the PhD degree in computer science from Aachen University of Technology, Germany. He is currently a professor of software engineering at the Blekinge Institute of Technology (BTH), Sweden. He is a member of SERL-Sweden, the Software Engineering Research Lab at BTH. His main research interests include empirical software engineering and cover requirements engineering, object-oriented methods, software process improvement, software measurement, software comprehension, and computer science education. He is a founding member of the Scandinavian Pedagogy of Programming Network and a senior member of the Swedish Requirements Engineering Network. He is a member of the IEEE.

Barbara Paech received the PhD degree in computer science from Ludwig Maximilians University Munich, Germany, in 1990, and a Habilitation in computer science from Technical University Munich in 1998. She holds the chair "Software Engineering" at Heidelberg University, Germany. Her teaching and research focuses on methods and processes to ensure quality of software with adequate effort. Since many years, she has been particularly active in the area of requirements and rational engineering. Based on her experiences as the department head at the Fraunhofer Institute for Experimental Software Engineering, her research is often empirical and in close cooperation with industry. She was spokeswoman of the section "Software Engineering" in the German Computer Science Society for six years and is founding member of the International Requirements Engineering Board. Since 2016, she has been the head of the advisory board of study affairs of the representation of German Computer Science study programs "Fakultätentag Informatik."