Noname manuscript No.
(will be inserted by the editor)

Commenting Source Code: Is It Worth It For Small Programming Tasks?

Sebastian Nielebock · Dariusz Krolikowski · Jacob Krüger · Thomas Leich · Frank Ortmeier

Received: October 10, 2018/ Accepted: date

Abstract Maintaining a program is a time-consuming and expensive task in software engineering. Consequently, several approaches have been proposed to improve the comprehensibility of source code. One of such approaches are *comments* in the code that enable developers to explain the program with their own words or predefined tags. Some empirical studies indicate benefits of comments in certain situations, while others find no benefits at all. Thus, the real effect of comments on software development remains uncertain. In this article, we describe an experiment in which 277 participants, mainly professional software developers, performed small programming tasks on differently commented code. Based on quantitative and qualitative feedback, we *i)* partly replicate previous studies, *ii)* investigate performances of differently experienced participants when confronted with varying types of comments, and *iii)* discuss the opinions of developers on comments. Our results indicate that comments seem to be considered more important in previous studies and by our participants than they are for small programming tasks. While other mechanisms, such as proper identifiers, are considered more helpful by our participants, they also emphasize the necessity of comments in certain situations.

Keywords Comments · Program Comprehension · Empirical Study · Documentation · Maintenance

Sebastian Nielebock · Dariusz Krolikowski
Otto-von-Guericke-University Magdeburg
E-mail: sebastian.nielebock@ovgu.de
E-mail: dariusz.krolikowski@darekkay.com

Jacob Krüger
Otto-von-Guericke-University Magdeburg & Harz University of Applied Sciences Wernigerode
E-mail: jacob.krueger@ovgu.de

Thomas Leich
Harz University of Applied Sciences Wernigerode & Metop GmbH Magdeburg
E-mail: tleich@hs-harz.de

Frank Ortmeier
Otto-von-Guericke-University Magdeburg
E-mail: frank.ortmeier@ovgu.de

1 Introduction

Developers spend most of their time maintaining, understanding, and familiarizing with existing source code (Standish 1984; Tiarks 2011; Siegmund 2016; Krüger et al 2018). Consequently, maintenance – comprising, for instance, bug fixing and updating – is often the most expensive phase of software development (Boehm 1981; Standish 1984; Chikofsky and Cross 1990; Sharon 1996). Improving the comprehension of a program reduces the necessary time for maintenance and the probability of introducing new bugs (von Mayrhauser and Vans 1995; Storey et al 1997). In particular, *program comprehension* is a research area that investigates how developers understand existing programs (Koenemann and Robertson 1991). Several patterns evolved to improve a program's source code and, thus, its comprehension, for example, guidelines for clean code (Martin 2009) or design patterns (Gamma et al 1995). However, these often emerge from personal preferences and experiences rather than scientific methods.

To address this issue, researchers conduct studies and propose approaches to investigate program comprehension (von Mayrhauser and Vans 1995; Storey 2005; Siegmund 2016; Schröter et al 2017). Several works address categories such as comprehending source code itself, for example, the importance of identifiers (Takang et al 1996; Anquetil and Lethbridge 1998; Lawrie et al 2007; Hofmeister et al 2017), or a program's behavior (Cornelissen et al 2007; Beck et al 2013; Kobayashi et al 2013; Trumper et al 2013). According to a recent study, fewer researchers seem to investigate the effect of documentation, such as comments, on program comprehension (Schröter et al 2017). Furthermore, as we discuss in Section 2, studies on comments contradict each other, rely mainly on students, or are older than 20 years, wherefore they do not use modern languages or paradigms. Thus, the real effect of comments on comprehensibility, especially with modern programming methods, remains uncertain.

Comments are a standard in most programming languages, became more powerful (e.g., with JavaDoc (Kramer 1999)), and enable developers to explain the code in their own words (Elshoff and Marcotty 1982; Corazza et al 2015). Consequently, they provide an additional mechanism to improve the comprehensibility of code. For this article, we differentiate three types of commented code, precisely: *Non-commented* code, code with *implementation* comments, and code with *documentation* comments. We explain their differences in Section 2.

In this article, we describe an empirical study (Section 3) based on small programming tasks, in which we investigate how these differently commented code types impact the comprehensibility of the source code. Even though the task sizes were mainly designed to motivate many programmers to participate in our study, these tasks can provide meaningful insights: First, maintenance tasks usually comprise a small fraction of code that developers need to comprehend and change in order to fulfill their task. For example, bug fixes typically consist of small code changes (Martinez and Monperrus 2015).

Second, our results provide initial insights for what code size comments are helpful. During programming, a developer may wonder whether the actual code is worth – more precisely, is large enough – to be commented. Thus, by considering small programs, we provide a first keystone to decide whether and which comments support comprehension of such elementary code parts (i.e., single methods with less than 30 source lines of code).

Finally, we analyze if there are differences between novice and professional programmers. Within this article, *professionals* represent experienced software developers, while *novices* are still learning to program, for example, first-year students. We describe our distinction between these two groups in Section 3.2.

In detail, we are concerned with the following three research questions:

- RQ_1 Is there a significant difference between the types of commented code and the correctness or time of task solving, respectively?
- RQ_2 Is there a significant difference between novice and professional programmers in the correctness or time of task solving for differently commented code types?
- RQ_3 Are the programmers' self-assessments of comments coherent with the observed results?

We analyze the first two questions based on an experiment, while we investigate the third question based on a questionnaire. Furthermore, we discuss the results and potential threats to validity in Section 4 and Section 5, respectively. Thus, our two main contributions in this article are the following:

First, we describe a **quantitative** experiment that we conducted as an online survey. For this experiment, we designed different programming tasks based on small programs – applying existing code, extending code, and fixing bugs – and three different kinds of commented code – non-commented code, code with implementation comments, and code with documentation comments. We measured the performance of the participants based on the proportion of correctly solved tasks and the time to do so. Overall, we received 277 responses with a high ratio ($\approx 81\%$) of professionals. Thus, we quantify the impact of different types of commented code on these differently experienced developers.

Second, we report the **qualitative** feedback of 157 participants from our online survey. We compare and discuss the obtained results with additional personal opinions, experiences, and responses of 86 participants. Based on this, we gain further insights into the usefulness and usage of comments, especially in industrial settings, as well as the participants' capabilities on self-reflection. Overall, we significantly extend the scope of current studies by considering a larger number of professionals and comparing differently commented code types using quantitative and qualitative methods.

2 Commenting Source Code

In this section, we provide a brief introduction on commenting source code. Afterward, we describe existing experiments to summarize the state of the art in this area and compare those to our work. Finally, we provide an overview of additional related work, for example, studies not considering comments explicitly.

2.1 Comments

While source code is mainly intended to be executed by a computer, it is important for developers to understand it (Knuth 1984; Standish 1984; Tiarks 2011;

```
1  /**
2  * Calculate the sum of all roman numerals
3  * in an array as an integer
4  * @param strings - array with roman numerals
5  * @return - sum of roman numerals as integer
6  */
7  public int foo(String[] strings) {
8    // variable to accumulate converted roman numerals
9    int number = 0;
10
11   // iterate over all roman numerals
12   for(int i = 0; i < strings.length; i++){
13       // add current roman numeral to result
14       number += convertRoman2Int(strings[i]);
15     }
16   return number;
17 }
```

Listing 1: Implementation and documentation comments.

Siegmund 2016). Useful comments may improve the comprehension, due to additional information (Elshoff and Marcotty 1982; Corazza et al 2015). Nonetheless, new code is rarely commented and existing comments are not updated with the code, for instance, because of time limitations, missing motivation, or automated refactoring (Jiang and Hassan 2006; Fluri et al 2007; Sommerlad et al 2008). To tackle this problem, approaches to detect legacy comments have been proposed in recent works (Tan et al 2012; Sridhara 2016; Ratol and Robillard 2017). Still, the overall effect of comments on comprehensibility is not obvious and requires empirical analysis.

In our experiment, we rely on Java, for which we can provide code snippets with three differently commented code types. We use the following terms, according to Vermeulen (2000) and the Java Code Conventions[1]:

- *No comments* (N) refers to uncommented source code. Thus, there is no additional documentation in the code.
- *Implementation comments* (I) describe one or multiple lines of code, for example, the implemented behavior, used algorithms, or known bugs. Such comments usually comprise a single line, as we show in Listing 1 in lines 8, 11, and 13. Their beginning is marked with `//` and prohibits further source code after the comment in that line.
- *Documentation comments* (D) describe mainly constructs, such as interfaces, methods, or classes. Usually, they only report information necessary to understand and execute these constructs, but not their concrete implementation. In Listing 1, we show the most common form of these comments in Java: JavaDoc (Kramer 1999) from line 1 to 6. JavaDoc comments became a powerful tool to comment source code, for example, by serving as input data to automatically create API documentation (Khamis et al 2010). In particular, these comments utilize block tags, such as `@param` or `@return`.

In our experiment, we use implementation comments for implementation details and JavaDoc for documentation details. There exist also other classifications, for

[1] `http://www.oracle.com/technetwork/articles/javase/codeconvtoc-136057.html`

example, based on the position of comments (i.e., leading, trailing, and freestanding (Sommerlad et al 2008)) or the reason for a comment (e.g., copyright, ToDo notes, or section marks (Martin 2009; Steidl et al 2013)). However, we are focusing on whether a comment explains the functionality of a whole method or of specific statements in a method. For this reason, we rely on the distinction between implementation and documentation comments. Other aspects like the position or finer granularity may also affect the comprehensibility, but are not part of this study.

2.2 Related Studies on the Effect of Comments on Program Comprehension

In recent decades, several studies on the influence of source code elements on program comprehensibility have been published. Within this section, we describe previous experiments that considered comments in the source code. As we partly replicate existing studies, we follow the suggestions of Carver (2010) on reporting replications. For this purpose, we provide detailed information about the identified studies, motivate our study, and clarify what parts we replicate and why. As we did not interact with any author of the previous studies and are not strictly following a previously used setup, we report our detailed study design in Section 3. We compare and discuss the findings of the identified studies and our own results in Section 4.4.

To identify existing studies, we applied a lightweight systematic review of the available literature. We started with an automated search in four digital libraries that index publications of important publishers in software engineering, namely DBLP, SCOPUS, Google Scholar, and the ACM Guide to Computing Literature. In these libraries, we searched all documents for which the **title** applies to the following search string (last checked July 18th 2018):

```
comment AND comprehension
```

This way, we identified an initial set of three studies:

- Woodfield et al (1981)
- Salviulo and Scanniello (2014)
- Börstler and Paech (2016)

Afterwards, we applied backwards and forwards snowballing (Jalali and Wohlin 2012; Wohlin 2014) on these studies using Google Scholar to extend the scope of our review (last checked July 18th 2018). With this procedure, we aim to reduce potential threats to the completeness that may appear, due to issues with searching in digital libraries (Jalali and Wohlin 2012; Shakeel et al 2018). We only selected documents that comprise an empirical study on the influence of source code comments on program comprehension. Thus, we finally identified ten empirical studies. In the following, we briefly describe each study and summarize the key details in Table 1. Afterward, we emphasize the need for extensions and match previous works to our own research.

Information about Existing Studies: Sheppard et al (1978) investigated the influence of different program characteristics, such as comments and structure. In this experiment, 36 professional programmers had to modify differently structured

Table 1: Summarized details of related studies and their comparison to this study.

Study	# Part.		Com.	Lang.	Meas.	Improvement
	Nov.	Prof.				
Sheppard et al (1978)	0	36	N, I, D	Fortran	T	No
Woodfield et al (1981)	48	0	N, D	Fortran	Q	For modularized code
Norcio (1982)	130	0	N, I, D	Fortran	C	For unindented code
Dunsmore (1985)	31	0	N, I	Fortran	C	Yes
	48	0	N, D	Fortran	—	Yes
Tenny (1985)	81	0	N, I	PL/I	Q	Marginally significant
Tenny (1988)	148	0	N, D	PL/I	Q	For monolithic code
Takang et al (1996)	89	0	N, D	Modula-2	Q, S	Yes
Nurvitadhi et al (2003)	103	0	N, D	Java	Q	Yes
Salviulo and Scanniello (2014)	18	12	D	Java	Q	Professionals no, novices yes
Börstler and Paech (2016)	104	0	N, I	Java	Q, S	No
This study	50	227	N, I, D	Java	T, C, S	-

T: Time; Q: Answering questions; C: Completing tasks correctly; S: Subjective opinion
N: No comments; I: Implementation comments; D: Documentation comments

and commented code fragments in Fortran. The authors found only performance differences when changing the structure, but not the comments of the code.

In another study, Woodfield et al (1981) distributed 48 students into two groups. One group received Fortran code with documentation comments, while the other group received uncommented code. Within a given time frame, the participants of both groups had to answer the same comprehension questions regarding the code. Participants who received modularized, commented code were able to correctly answer more questions than those who received the same, but uncommented code. In contrast, for monolithically structured code, no significant differences were found.

Norcio (1982) investigated the effects of indentation and comments. For this purpose, 130 students participated in two experiments and had to correctly complete different versions of Fortran programs. The results indicate that comments have a significant positive impact on comprehension when no indentation is used.

Dunsmore (1985) conducted two experiments on the effect of implementation and documentation comments. The studies relied on differently sized Fortran code for which different numbers of students, 31 and 48 respectively, had to perform specific tasks. Unfortunately, the author reports few details about the studies (e.g., it is not specified what measurements were used for the second experiment and the subject code is not provided), but they seem to be connected to the one of Woodfield et al (1981). For both experiments, Dunsmore (1985) reports a positive effect of comments on program comprehension and modification.

Tenny (1985, 1988) analyzed the influence of comments and procedure types on comprehension in two different experiments. Both were conducted with students, 81 and 148 respectively, and different variants of PL/I systems. Each variant was either commented for each code section and procedure or did not contain any comments. Regardless of the type of the procedure, in both experiments, students were able to answer more questions correctly if they read the commented code. However, the results indicated that comments were marginally significant and significant only in a monolithic program structure, for each of the two experiments.

Takang et al (1996) analyzed the impact of comments and identifier names. In their study, 89 computer science students had to answer questions on Modula-2 source code and had to assess the program readability. While in the questionnaire

comments tended to improve readability, this was not supported by the participants' subjective assessment. The authors conclude that comments, as well as identifier names, improve the understandability of programs.

In a study on the influence of Java documentation comments by Nurvitadhi et al (2003), 103 students had to answer a questionnaire. The authors considered class and method comments on documentation level, exclusively and together, as well as without comments. Their findings are that students who received both types of comments had significantly better scores than the other groups. For low-level questions, method comments improved code comprehension in comparison to uncommented code.

Salviulo and Scanniello (2014) analyzed the influence of identifiers and comments with 18 students and 12 professional developers. Within their controlled experiment, the participants had to answer questions and accomplish different tasks on a medium sized game implementation in Java. The authors discuss that professional programmers tend to ignore comments while solving their tasks. In contrast, students emphasize the importance of comments.

Finally, Börstler and Paech (2016) performed a study on comments and method chains. They used data from 104 students who assessed different versions of a Java system that contained no, "good", or "bad" comments. The authors used cloze tests and subjective assessments to evaluate the readability of the code. They found no significant differences for any comment type. However, "good" comments were considered best and no comments as worst readable.

Why another study? We have four main reasons that motivate this study. Based on these, we argue that our work provides significant value to the research community and practitioners alike.

Firstly, replications in empirical software engineering help to validate and consolidate existing knowledge (Basili et al 1999; Juristo and Vegas 2009; Bezerra et al 2015). Thus, several authors, for example, Nurvitadhi et al (2003) and Salviulo and Scanniello (2014), of the described studies themselves emphasize the importance of replicating their experiments. Considering the varying and partly contradicting findings of previous studies, further replications seem necessary to investigate the suitability of comments for documentation purposes.

Secondly, we see threats to the validity of existing studies in modern and especially industrial settings. This is mainly due to the fact that most studies are older than 20 years (Sheppard et al 1978; Woodfield et al 1981; Norcio 1982; Dunsmore 1985; Tenny 1985, 1988; Takang et al 1996). They use older programming languages and paradigms (e.g., Fortran), and rely mostly or solely on students (Woodfield et al 1981; Norcio 1982; Dunsmore 1985; Tenny 1985, 1988; Takang et al 1996; Nurvitadhi et al 2003; Salviulo and Scanniello 2014; Börstler and Paech 2016), who may not be representative for real-world evaluations (Höst et al 2000; Runeson 2003; Svahnberg et al 2008). In contrast, for our experiment, we use Java, which is widely used today, and not only 50 novices but also 227 professional programmers participated in our study. We remark that only 66 of the participants finished the study completely, wherefore some tasks received fewer responses. In Section 3.1, we describe this issue in detail.

Thirdly, we find that several studies consider additional aspects of a program, such as modularity (e.g., Woodfield et al (1981)) or identifier names (e.g., Salviulo and Scanniello (2014)). Due to dependencies between these aspects, analyzing the

actual effect of comments separately may be difficult. This threatens the internal validity of these studies (Perry et al 2000; Siegmund et al 2015). Furthermore, the used measurements are mainly subjective, providing rarely quantitative results. In contrast, we conduct our experiment focusing only on the effects of comments and combine quantitative with qualitative measurements to gain detailed insights.

Fourthly, most studies, except for Salviulo and Scanniello (2014) partly covering $\mathbf{RQ_2}$, address solely $\mathbf{RQ_1}$ with some of the previously mentioned limitations. Thus, our other research questions are scratched at best and reliable results are missing. For this reason, we are not only replicating previous studies, but extend their scope to provide more insights. In particular, we consider differences between novices and professionals as well as subjective opinions on comments compared to their actually measurable impact.

2.3 Related Work on Comment Analysis and Program Comprehension

There are several other works that investigate comments for different purposes and empirical studies on program comprehension. In the following, we provide a brief overview of some of them. The described studies complement our work by investigating the effects of other aspects on program comprehension. Furthermore, the results of our study can be used as a basis for scoping and extending the described approaches that utilize comments.

A recently proposed approach is to automatically generate comments that are considered helpful in understanding source code (Wong et al 2013; McBurney and McMillan 2014). The question arises, by which properties "good" comments are characterized. To this point, McBurney and McMillan (2016) conducted an experiment and found that comments written by the authors usually use keywords from the source code. A reason could be that comments with these keywords directly provide relations to the code. Moreover, the authors determined that the similarity of human-written comments and code can be measured with text similarity metrics, while it cannot be measured between generated comments and code. Similarly, Buse and Weimer (2010) investigate the readability of code comments and develop a corresponding measure. Furthermore, the authors show that their measure correlates with three other quality measures: Code changes, automated defect reports, as well as defect log messages.

Program comprehension is not solely influenced by comments. Several studies analyze the influence of different artifacts, approaches, and human factors, for example, software design techniques (Briand et al 1997), the application of domain specific languages instead of general purpose languages (Kosar et al 2012), visual code highlighting (Feigenspan et al 2013), static typing (Hanenberg et al 2014), code repetition (Jbara and Feitelson 2015), identifier names (Hofmeister et al 2017), or developers' memory (Krüger et al 2018). Most approaches recognize a significant positive or negative influence on program comprehension. Thus, we have to keep in mind that other aspects of software development can be more important than commenting code to improve the understandability. As a result, many different approaches have been proposed to improve or investigate the understandability of code.

Other studies analyze the application of comments in other contexts than program comprehension. For instance, Ying et al (2005) detect that programmers

use comments also for internal communication, for example by applying ToDo-comments. Ali et al (2015) investigated the effect of comments on requirements traceability and found that they have a significant impact. Several authors (Ji et al 2015; Seiler and Paech 2017; Krüger et al 2018) use annotations in a comment-like style to integrate feature traceability in the source code, emphasizing their benefits. Thus, comments may also have an influence on traceability, which was found to have positive effects on programming (Mäder and Egyed 2015). To this end, several authors proposed techniques to automatically trace documentation to the code and to use this traceability for different purposes. For example, Antoniol et al (2002) describe a technique to automatically recover traceability links between code and documentations by analyzing identifier names. Moreover, Sridhara et al (2010) propose a technique to automatically generate summary comments for Java methods to provide up-to-date documentation in natural language.

3 Design of the Online Survey

The goal of our study is to ascertain the influence of differently commented code types on correctness and time of solving small programming tasks. In particular, we varied the types of commented code among groups of participants and measured the time it took them to correctly solve each task. For this purpose, we conducted an online survey between June 1st and July 11th 2016. While an online survey does not allow us to observe our participants as good as a controlled experiment, such methods have several benefits considering the diversity of participants – preventing biases – and prove to be consistent with traditional methods (Gosling et al 2004). Before conducting the study, we tested it with five participants and rectified it if necessary. In particular, we aimed to improve the quality of our examples, for instance, if code or comments were hard to understand or ambiguous. These five participants were not part of the final study.

The survey was completely conducted in German and mostly targeted native speakers to avoid language barriers. Particularly, tasks and comments were provided in German. However, for the sake of replicability and repeatability, we translated the tasks and comments. All tasks and our basic solutions can be found in Appendix A of this article.

Technologically, we deployed the open source LimeSurvey.[2] To facilitate sampling and to avoid selection biases, we extended this tool in order to automatically distribute participants to different groups. We provide an overview of the main factors of our survey in Table 2 – similar to the scheme by Hofmeister et al (2017) – and describe our procedure in the following.

3.1 Acquisition and Data Rectification

We promoted our study in an academic as well as in an industrial context via mail and social media. Overall, 416 participants started the survey. Due to our decision to perform an online survey, we consciously designed an unsupervised experiment. This means, even though the programmers were informed not to disturb

[2] `https://www.limesurvey.org/`

Table 2: Main factors of the conducted online survey.

Goal	Study the impact of differently commented code types on program comprehension, precisely, correctness and solving time of programming tasks.
Independent Variables	Commented code types, programming experience
Tasks	Apply existing code, fix bugs, extend code
Dependent Variables	Correctness, time
Secondary Factors	Influence of comments on different programming tasks
Confounding Factors	Materials (code snippets), identifier names, interruption/abandonment of the study, inter-individual differences, item-order
Design	Within-subjects

their work, some participants were interrupted or did not finish the experiment at all. Thus, we removed those participants from the initial 416, who answered the preliminary questionnaire, but did not proceed to the programming tasks or whose task execution times appear to be unrealistic (see below), resulting in 277 participants. Among these 277 participants were 227 professional and 50 novice programmers, based on our classification described in the following section.

We excluded those answers that exhibit an unrealistic execution time, meaning either too long or too short times. Precisely, we consider a time as too long, if it is greater than the third quartile plus the threefold interquartile range (difference of the first and third quartile) of all submissions of the task in the particular group, regardless of the participant's experience. In contrast, as the same mechanism does not work for too short answer times, due to the lower bound of 0, we removed those solutions that had an answer time of fewer than ten seconds. Furthermore, some participants mentioned in the post-questionnaire disturbances while solving specific tasks, so that we removed these results, regardless of statistical deviation. Therefore, not all programming tasks have the same number of participants. By means of this procedure, we omitted 321 single tasks from the original obtained 1,940 tasks, leaving 1,619 tasks from 277 different participants. For 66 (7 novices and 59 professionals) participants, we could keep all obtained tasks. Still, for each individual task, we received more than 100 responses (ranging from 127 to 262) as basis for our analysis (cf. Figure 1 in Section 4).

3.2 Structure of the Study

Our study comprised three steps: First, we assessed programming experiences to distribute participants among different groups. Second, each group had to solve nine programming tasks with differently commented code. Finally, the participants had to self-assess the influence of comments and got the chance to give feedback. We describe the details of these steps in the following paragraphs.

Assessment of Programming Experience The assessment of programming experience is a widespread domain, and there exists no standard metric to be measured, as found by Feigenspan et al (2012). In their study, the authors analyzed different methods to measure programming experience including, for example, years of

Table 3: Categories to quantify programming experience.

Experience Value	1	2	3	4	5	6
Self-Assessment	1	2	3	4	5	6
Years in Programming	<1	1-3	3-6	6-10	10-15	>15
Qualification in Programming	-	no	-	-	yes	-

programming, educational background, and self-assessment. They developed a five-factor model, which encompasses different measurements and enables researchers to assess programming experience.

We adapted this idea and asked the participants to self-assess their programming skills, state their years as programmers, and whether they have a qualification in programming or not. As we performed an automatic assignment to the groups, we defined for every category a number between 1 and 6, as we depict in Table 3. These numbers represent experience values where 1 means low and 6 high experience. Note, as the question about the qualification is binary (yes or no), we assigned the mean value of the lower and upper part of the experience value (i.e., 2 and 5), respectively. The scale for years in programming is inspired by one of the largest studies with programmers performed by Stack Overflow,[3] one of the most common Q&A systems for programmers, and adapted to fit into our scale.

In the end, we calculated the rounded average of these three factors as experience value. We consider participants as novices, if their experience value is ≤ 3, and as professionals if the value is ≥ 4. Note that this assessment allows students to be categorized as professionals, even though our procedure takes the participant's qualification into account. Nonetheless, some students have been working in the industry for several years and, thus, can be considered as experienced developers.

Programming Tasks To measure the influence of differently commented code types, our participants had to process three different maintenance aspects – *applying existing code* in the program, *fixing bugs*, and *extending the code*. *Applying existing code* means that a programmer should use the API of the provided code, for instance, its methods, in the correct manner in order to fulfill a given task. When participants should *fix a bug*, they have to repair a negative behavior in the code, for instance, an exception or a non-intended output, in the given code. Finally, *extending the code* means adding a new feature or a new functionality to the existing code. For instance, participants have to add a parameter to the existing methods to provide a new functionality. Each of these aspects comprised three programming tasks with differently commented code. We distributed the participants into three distinct groups for which we varied the commented code type in each task, as we display in Table 4. This way, we overcome the problem that differences in time are mainly dependent on the particular programming task or on inter-individual differences of the participants.

At the beginning of our study, the participants did not know that we investigate the influence of differently commented code, in order to not bias our results. Furthermore, they were allowed to use their own programming environment (i.e., usual IDE), to not influence the programming time, due to an unfamiliar

[3] `https://insights.stackoverflow.com/survey/2016#developer-profile-experience`

Table 4: Mapping of groups and programming tasks with no (N), implementation (I), and documentation comments (D).

Aspect	Task	Group		
		A	B	C
Apply code	1	N	I	D
	2	I	D	N
	3	D	N	I
Fix bug	4	I	D	N
	5	D	N	I
	6	N	I	D
Extend code	7	D	N	I
	8	N	I	D
	9	I	D	N

environment (Siegmund 2016). To diminish effects of identifiers on program comprehension (Takang et al 1996; Anquetil and Lethbridge 1998; Lawrie et al 2007; Hofmeister et al 2017), we used anonymous classes (e.g., Class1), methods (e.g., foo()), and variable names (e.g. string1, number). Other effects, such as code indentation or keyword highlighting, were preserved and equal for every participant.

We designed the programming tasks to be short enough for the participants to process the whole study in less than an hour. Moreover, we only used Java to prevent the overhead of acclimating to other languages during the study. According to various indexes,[4] Java is still one of the most popular programming languages. As mentioned before, the tasks were short (i.e., 7 to 27 source lines of code) and mainly addressed algorithmic problems. However, we based some of our samples on existing open-source projects, namely the Apache Common Lang project and Guava. Particularly, we used code of Apache Common Lang for task 6 – where we inserted a bug into an existing method – and for task 8 – where we ask for an adaptation of another method.

Within the code snippets, we manually inserted the two types of comments, implementation and documentation, that we describe in Section 2. We did not consider the use of existing comments for two reasons: First, for those samples we partially derived from existing projects, we cannot ensure the correctness of the existing comments. Since we designed the tasks, which are mainly of an algorithmic nature, we feel very confident that the comments are correct and valuable. We also checked that by testing our study with a preliminary run of the study with five participants and rectified our comments if necessary. Second, most of the considered code snippets do not contain both, implementation and JavaDoc comments, sufficiently and, thus, we need to write the comments on our own.

Due to the small size of the snippets, comment generators (McBurney and McMillan 2014; Rahman et al 2015) are not applicable, as they mainly benefit from the source code's context or an external source. For documentation comments, we used the JavaDoc syntax and described the purpose of the entire class as well as its methods. This also encompasses a description of the purpose of input as

[4] TIOBE: `https://www.tiobe.com/tiobe-index/`
RedMonk: `http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/`
PopularitY: `http://pypl.github.io/PYPL.html`

well as return values with respect to the particular method. In contrast, we used implementation comments to explain the purpose of a statement, for example, the meaning of a branch statement or a variable assignment at a particular location. We obtained the non-commented variant by removing all existing comments. Note that we created the comments in such a way that they do not just repeat the code syntax and that they do not contradict the semantics of the code.

Post-Questionnaire Finally, we asked the participants to self-assess the effect of documentation and implementation comments on their response time in comparison to non-commented code. To this end, we utilized a Likert scale (Trochim et al 2016) with five possibilities from significantly slower to significantly faster, containing the possibility that the time could be identical (i.e., no effect of differently commented code types). Additionally, the participants had the opportunity to give insights into their used auxiliary material and further feedback as free-text.

4 Results of the Online Study

In this section, we analyze the outcome of our study. We address each research question by describing the corresponding results and discussing their implications.

4.1 RQ_1 – Effect of Different Comment Types

First, we investigate the differences between differently commented source code. In particular, we are interested in the influence of comments on the correctness, precisely, whether the participants were able to correctly solve the tasks, and, for the correct solutions, the impact on task execution times. To assess correctness, one of the authors created a sample solution for every task, which served as a loose specification to check submissions. This author manually assessed for every submission if it behaves in the same way as the sample solution. Note that solutions that solve the task were also accepted, even if they differ from the sample solution. All experiments are conducted under the *null hypothesis* that there are no differences between the differently commented code types for every single group (i.e., novices and professionals). Thus, our analysis considers whether this hypothesis holds or not. If it has to be rejected, we provide a post-hoc analysis to reveal between which types of code significant differences exist.

Results Unsurprisingly, we obtained different proportions of correct and wrong submissions, which we depict in Figure 1. We illustrate the correctness based on the types of commented code – N (non-commented code), I (code with implementation comments) and D (code with documentation comments) – and, with respect to **RQ_2**, on the different experience levels – novices and professionals. In addition to the proportions, we also show the absolute numbers. Note that due to our rectification process, the number of participants varies per task and group.

With respect to the correctness of answers, we analyze the differences of comment types separately for both experience groups. Due to the small number of novice participants, we use Fisher's exact test ($\alpha = 0.05$) (Fisher 1936) to the χ^2-test. If Fisher's exact test found significant differences between the three differently

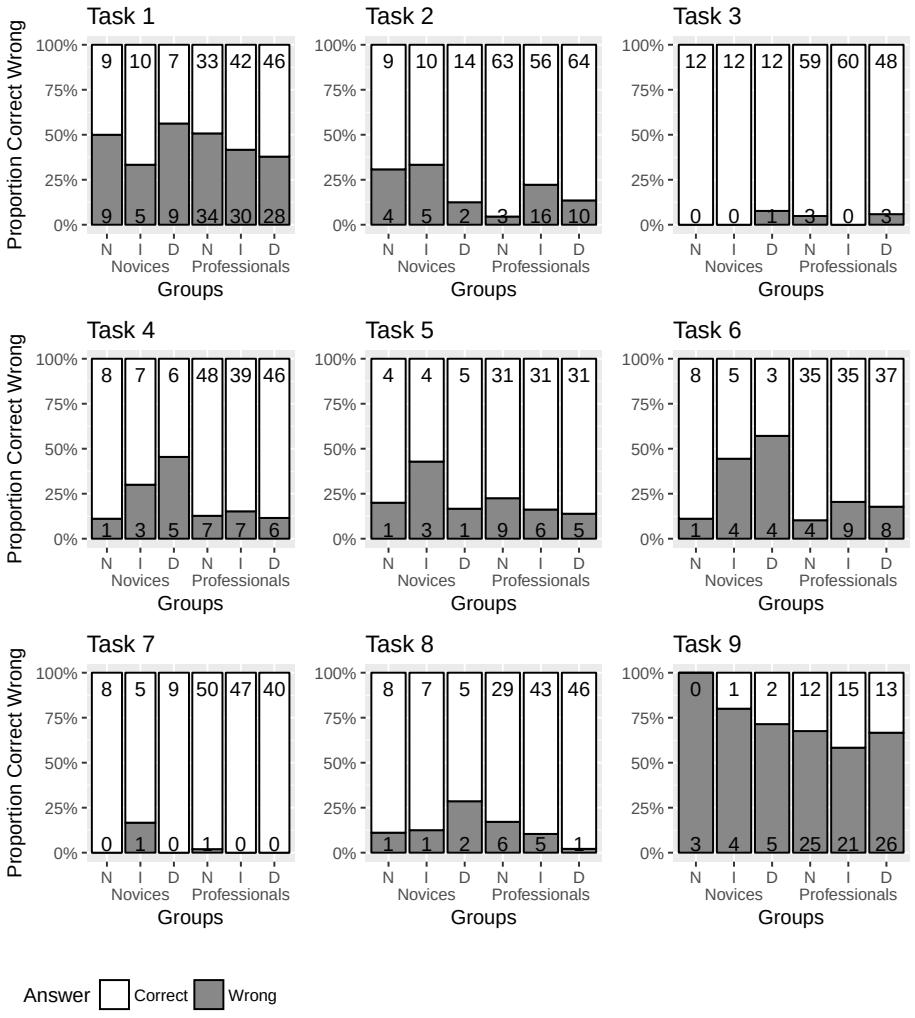


Fig. 1: Correctness for each task (N: No comments; I: Implementation comments; D: Documentation comments).

commented code types, we conducted Fisher's exact test as a pairwise post-hoc analysis. In order to deal with family-wise error rate due to multiple statistical tests, we applied the Bonferroni adaption of the p-value (Dinno 2015). We depict all results in Table 5.

As we show in Figure 1, the proportions and absolute numbers indicate more variation in the correctness for novices. However, Fisher's test does not reveal any significant differences for this group. Within the group of professionals, only the difference in the correctness of task 2 is significant. Interestingly, this difference is between *non-commented code (N)* and *code with implementation comments (I)* (right side in Table 5). Thus, more professionals were able to solve task 2 correctly,

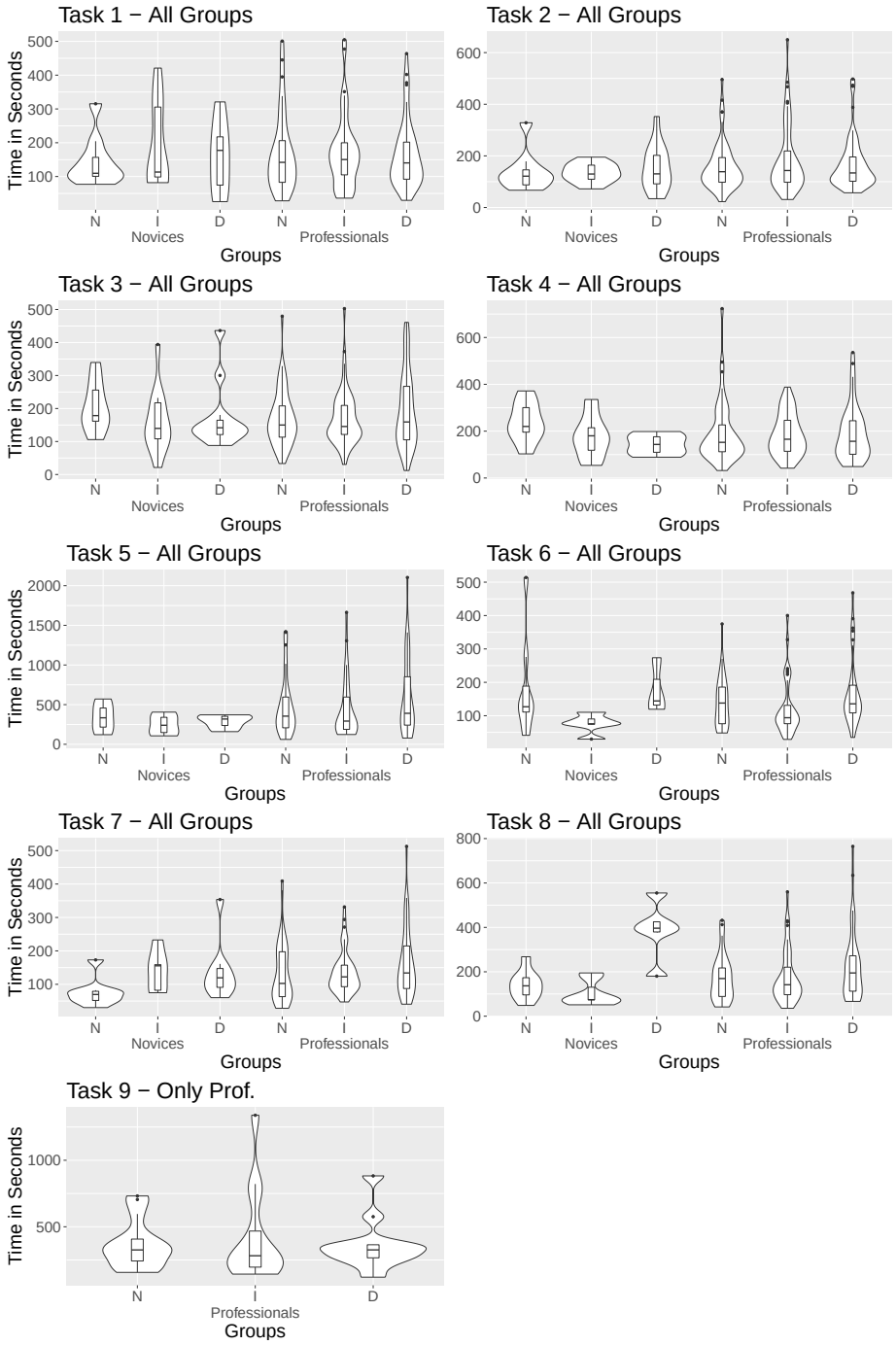
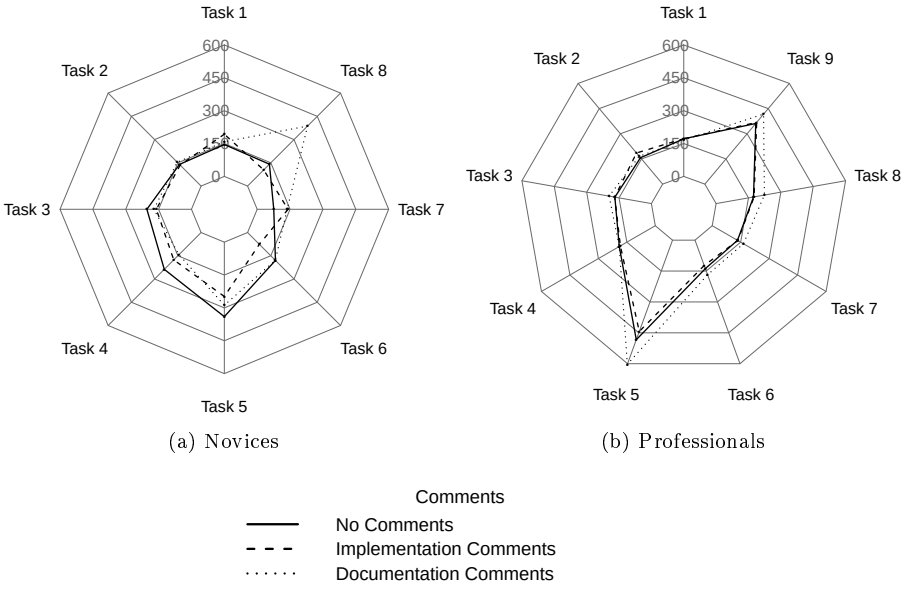


Fig. 2: Distribution of task answer times for correct solutions regarding experience and commented code types (N: No comments; I: Implementation comments; D: Documentation comments).

Table 5: Results for correctness using Fisher's exact test (left) and post-hoc test for task 2 - Professionals (right).

Task	Nov.	Prof.
1	0.4246	0.2747
2	0.3833	**0.0085**
3	1.0000	0.1612
4	0.3225	0.8720
5	0.6691	0.6130
6	0.1823	0.4385
7	0.2609	1.0000
8	0.6443	0.0598
9	1.0000	0.6648

Comments	p-value for Task 2 Prof.
N – I	**0.0082**
N – D	0.2523
I – D	0.5938



(a) Novices (b) Professionals

Fig. 3: Comparison of the mean times in seconds between differently commented code types.

if the code was not commented. This may happen if programmers are confused by the comments or misunderstand them.

Among all tasks and participants, we cannot see a significant influence of the differently commented code types on the correctness at all. Instead, Figure 1 indicates that the proportions of correct and wrong answers are mainly driven by the tasks themselves. For example, tasks 3 and 7 reveal a high ratio of correct answers, while task 9 has a significantly lower one. This may be the result of varying difficulty levels or participants' different comprehension of the tasks.

Next, we consider the differences in the answer times among differently commented code of correctly solved tasks. For that purpose, we represent the times of

Table 6: Results for execution times with Kruskal-Wallis test (left) and Dunn-test for task 8 - Novices (right).

Task	Nov.	Prof.
1	0.7884	0.9484
2	0.7952	0.7807
3	0.1699	0.6889
4	0.0985	0.9032
5	0.6623	0.3258
6	0.0787	0.0509
7	0.0568	0.2151
8	**0.0082**	0.2203
9	————	0.7705

Comments	p-value - Task 8 Nov.
N - I	0.6936
N - D	**0.0235**
I - D	**0.0040**

correct submissions with respect to the type of commented code and experience level as violin plots in Figure 2. Note that we do not depict the results of task 9 for novices since the number of correctly solved tasks is too low (3 out of 15 submissions among all types of commented code) and, thus, they are hardly representative. In order to facilitate a comparison, we depict the mean answer times for novices and professionals as radar charts in Figure 3.

Regarding the answer times, we see more variation in Figure 2 for novices compared to professionals. In particular, for task 8 with documentation comments, novices tend to require more time for correctly solving the task compared to the other types. We apply the Kruskal-Wallis test (Kruskal and Wallis 1952) with the null hypothesis of equal distributions ($\alpha = 0.05$) and ascertained the significance of these differences. Note that we have chosen Kruskal-Wallis, as this test does not require a normal distribution of the answer times. We show the results in Table 6, which illustrate that only the differences in task 8 for novices are significant. In order to assess between which types these differences exist, we conducted the Dunn test as a post-hoc analysis with conservative Bonferroni adaption (Dinno 2015), displayed on the right side of Table 6. As the violin plots in Figure 2 show, the differences between execution times of submissions with documentation comments is significantly higher than those of the other two code types for this task. Also, in the radar chart in Figure 3b we can see that the effect of comments for professionals is almost negligible. For them, we found no significant differences in any answer time of correct solutions.

We analyzed whether the particular content of a comment or further meta information, namely, the size of the comments, influence correctness and time. For that purpose, we consider the respective source lines of code (SLOC) – all lines without comments and blank lines – and the comment lines of code (CLOC) – all lines that contain comments.[5] Note that we obtain this information from the original source code with German comments, wherefore the numbers may slightly vary from the code presented in Appendix A. We summarize our results in Table 7. Since the number of tasks and, thus, the statistical significance is rather low, we only provide a qualitative discussion of the results. However, this discussion can motivate further studies that strive for statistical evidence of these results.

[5] We did not count lines such as `/**` or `**/` that do not contain any natural words.

Table 7: Size and content of comments compared to correctness and mean time.

T	S	Implementation Comments				Documentation Comments			
		C	Content	%	Sec.	C	Content	%	Sec.
1	15	4	Explaining values	70.0	137.67	7	Explaining purpose of the method and values of input/output variables	58.8	141.75
2	14	2	Explaining purpose of the code	79.6	178.16	7	Explaining purpose of the method and the validity of values with input/output variables	86.6	162.84
3	27	4	Naming variables and explaining purpose of code	100.0	168.26	6	Explaining purpose of the class, methods, and the validity of values	93.7	194.28
4	10	3	Naming variables and explaining purpose of code	84.1	189.61	4	Explaining purpose of the method with input/output variables	82.5	179.75
5	9	3	Explaining purpose of the variables and the code	82.0	410.57	5	Explaining purpose of the method based on definition with input/output variables	85.7	565.35
6	7	1	Explaining purpose of code	79.0	123.69	5	Explaining purpose of method with input/output variables	76.9	172.11
7	27	2	Explaining purpose of variables and code	98.1	135.56	3	Explaining purpose of method with input/output variables	100.0	159.53
8	21	3	Explaining purpose of variables and code	90.1	169.41	6	Explaining purpose of method with input/output variables	94.4	243.07
9	21	7	Explaining purpose of variables and code	59.6	378.34	7	Explaining purpose of class, methods with input/output variables	32.6	432.81

T: Task, S: SLOC, C: CLOC, %: Correctness in %, Sec.: Mean time in seconds

We can see in Table 7 that the number of CLOC for documentation comments is usually bigger than for implementation comments. In some cases, namely, tasks 5 and 9, this is correlated with a longer mean time to solve the tasks correctly. However, in task 5, the percentage of correctly solved tasks is slightly higher in the case of documentation comments, while in task 9 the percentage is smaller regarding implementation comments.

Another difference is the scope described within the comments. As explained before, documentation comments tend to describe more abstract concepts, for example, on class or method level, while implementation comments describe the purpose of specific lines. However, we do not see an indicator that these different concepts influence the correctness or time of any task, as all aspects seem to be positively as well as negatively correlated with each comment type.

For task 2, we found a significant difference in the correctness for professionals between non-commented code and code with implementation comments. Particularly, this difference shows a negative impact of implementation comments on this task. The corresponding 2 CLOC describe the concepts *prefix* and *suffix*. Due to the obfuscation of the variables, one may get confused, since the actual two string variables represent not prefix and suffix, but rather the input and output variables. More supportive comments may directly name the code parts representing the prefix and the suffix.

Novices needed more time to correctly solve task 8 with documentation comments, indicating a potentially negative impact. In this comment, we used the `@see` JavaDoc annotation to refer to the documentation of the `StringBuilder` class – a

class from the `java.lang` package. One reason for the increased answer time could be that novices try to familiarize themselves with that class. Professionals usually know this class and, thus, do not need to refer to its documentation.

Overall, the reasons for these significant differences in the two tasks are versatile. Consequently, a study with differently designed comments and combinations is necessary to reveal what reasons cause the negative effects. For example, such a study may vary the content, length, or expressiveness of the information provided in comments.

Discussion Our results reveal that significant variations in using different types of commented code are sporadic, which makes it hard to judge whether there exists a common pattern in these differences. Even though the correctness and answer times for novices tend to vary more than for professionals, we cannot find statistical evidence for most of these phenomena, partly because the number of correct submissions is too small. Overall, our findings indicate that both, correctness and answer times, are mostly influenced by the task itself and not by the type of comments used. We also find no general indicators whether a certain content or size is positively or negatively correlated with the correctness or time of solving tasks. Even though, we find some explanations why some tasks reveal significant differences. Still, we need to validate these with a much larger study of differently designed comments.

> Regarding **RQ_1**, our experiment shows that the effect of different comments on correctness and time is **almost negligible** for small programming tasks. Whether there exists differences for different and more complicated tasks or differently designed comment sizes and contents remains an open question.

4.2 RQ_2 – Novices versus Professionals

Even though the intra-group differences for the differently commented code types seem insignificant, inter-group differences among novices and professionals may occur. Similar to the previous research question, we examined the correctness and answer times of the tasks. In particular, we are interested in differences between both groups of participants that received the same type of commented code.

Corresponding to prior studies (cf. Section 2.2), we expect that novices benefit from comments, as they are less familiar with programming and can utilize additional information to comprehend a program. In contrast, professionals often ignore comments (Salviulo and Scanniello 2014), which is why we assume no improvements for them. Thus, we expect to observe more significant differences between novices and professionals when dealing with non-documented code than with the other kinds of commented code.

Results We analyze the differences in correctness once again with Fisher's exact test ($\alpha = 0.05$), whose results we depict in Table 8. Note that we do not perform a post-hoc test, as we are just comparing two groups, whose differences we can see in Figure 1. Our findings reveal significant differences between novices and professionals. However, only in task 2, we observe that professionals perform significantly better than novices when confronted with non-commented code. Remember that

Table 8: Results for correctness among novices and professionals with Fisher's exact test.

Task	N	I	D
1	1.0000	0.7731	0.2621
2	**0.0123**	0.3439	1.0000
3	1.0000	1.0000	1.0000
4	1.0000	0.3608	**0.0174**
5	1.0000	0.1383	1.0000
6	1.0000	0.1985	**0.0415**
7	1.0000	0.1132	1.0000
8	1.0000	1.0000	**0.0412**
9	0.5409	0.6317	1.0000

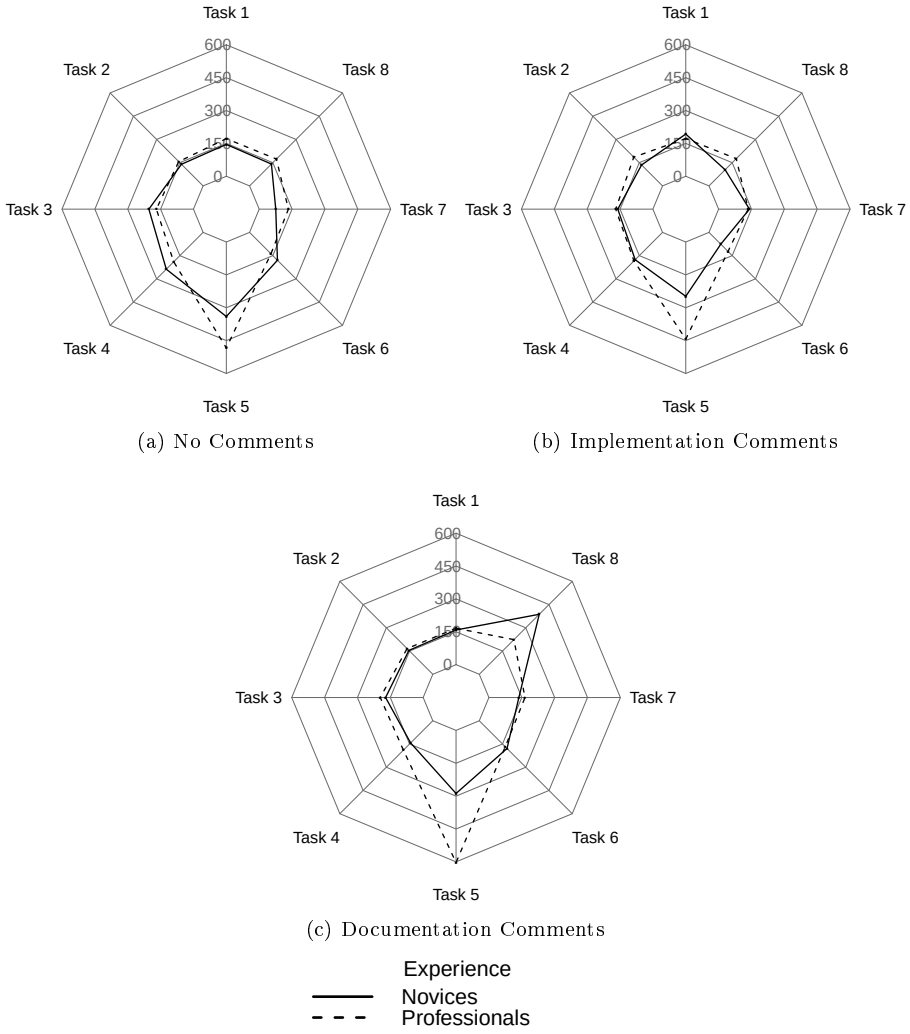
we also found a significant difference in professionals' correctness regarding the type of commented code for this task. Thus, in this particular task, professionals seem to perform especially well when no comments are provided.

In the tasks 4, 6, and 8, professionals performed better than novices, when confronted with documentation comments. There could be two possible explanations: Either, professionals, due to their experience, are superior to novices regarding interpreting documentation comments, or novices are more confused by documentation comments. As we find no significant differences in the correctness of novices' submissions among all comment types, we argue that the first option seems to be a more reasonable explanation. Moreover, the effect of improved correctness among documentation comments occurs frequently, which can reveal a common effect. In particular, the tasks 4 and 6 are bug fixing tasks, and, thus, these effects may be correlated with this aspect. However, bug fixing encompasses many variants of possible fixes, for example, syntactic versus semantic bugs, which we did not capture with our study.

Considering the meta-information of comments provided during these tasks (cf. Table 7), we cannot see a correlation between CLOC and correctness. Regarding the content of the comments, they briefly describe the purpose of the methods in tasks 4 and 6. Our results indicate that for *bug fixing* this description is not sufficient for novices. Considering task 8, novices may perform worse, due to `@see` annotations, as mentioned in the previous section. Note that these are only single observations and we highly recommend to validate these with further studies.

To analyze the effects on answer time, in addition to Figure 2, we also depict the individual differences of the mean times in single radar charts in Figure 4. Note that due to the low number of correct submissions by novices, we do not compare the results of task 9. For most tasks, the differences between the answer times of both groups are negligible. Interestingly, the mean time we measured for task 5 is consistently lower for novices compared to professionals.

In order to examine if these differences are significant, we analyze them via the Kruskal-Wallis test ($\alpha = 0.05$). The test reveals that there are no significant differences among the execution times of correct submissions for novices and professionals when confronted with the same type commented code. Thus, the observed improved performance by novices could be only by chance.



(a) No Comments

(b) Implementation Comments

(c) Documentation Comments

Fig. 4: Comparison of the mean times in seconds between novices and professionals for differently commented code types.

Discussion Our findings indicate that if professionals are confronted with documentation comments, they perform significantly better than novices. However, as this is not a regular phenomenon, it could be that these differences occur only for particular tasks, such as bug fixing. Because we conducted our experiments only with three tasks per programming aspect, we cannot give a final answer to this question, as bug fixing comprises various bug types. With respect to the answer time for correct submissions, we cannot find any significant differences between novices and programmers.

Regarding **RQ_2**, our experiment indicates that professionals perform **significantly better** in correctness than novices if they are confronted with **documentation comments**. Still, if this is by chance and which factors influence this performance has to be answered in future research.

4.3 RQ_3 – Programmers' Assessment

Finally, we asked the participants to subjectively evaluate the impact of comments. In particular, we were interested in the differences between their assessment of the impact of the differently commented code types and differences between novices and professionals. Note that the impact assessment is highly subjective. For instance, one developer may assess comments only as a highly decreasing factor when they save several hours of programming while another assesses them equally when saving a couple of minutes. Therefore, we consider these results as a measurement of how attractive or unattractive comments are. Moreover, we investigate whether the subjective evaluation is coherent with the findings of **RQ_1** and **RQ_2**. We expect that all participants assess comments as slightly beneficial.

Results As we show in Figure 5, most novices and professionals think that comments do not affect or slightly reduce the necessary time for maintenance. In detail, even though the bar plot visually indicates that novices tend to prefer implementation to documentation comments, a Wilcoxon-signed-rank test[6] (Siegel 1956) reveals no significant differences. Similarly, we cannot ascertain a difference in the distributions for professionals. Overall, both groups have a slightly positive expectation of comments: 14 to 11 answers of the novices and 132 to 37 answers of the professionals assessed comments as a moderately or highly time decreasing factor.

This is contrary to the results of the previous research questions, as they reveal that comments either have no significant effect or, for instance, in task 2 for professionals, programmers perform better without comments. Note that we have to be careful with the negative effect of comments, as this is not a commonly observed pattern within our results. Nonetheless, the results indicate that comments are seen as a positive support measure in software development. Thus, the importance and usefulness of comments regarding program comprehension may be higher than we measured, at least for small programs.

Considering this discrepancy, we also asked the participants to state their own opinions on the study and the topic, which was used by 86 of them. We summarize the responses that are concerned with the usefulness of comments in Table 9. We see that some participants are convinced that comments are only appropriate in specific situations. For example, some participants stated that using JavaDoc is problematic and only useful for interfaces. Other participants wrote that implementation comments are helpful in longer or complex source code. Thus, the aforementioned discrepancy may be a result of the tasks we designed. For our purpose of comparing differently commented types of code, we used small and not too complex examples. Based on these statements, additional studies on suitable usage scenarios for comments (e.g., documentation for interfaces) seem necessary.

[6] We applied this test since we want to compare two distributions from the same population, but we cannot assume a normal distribution.

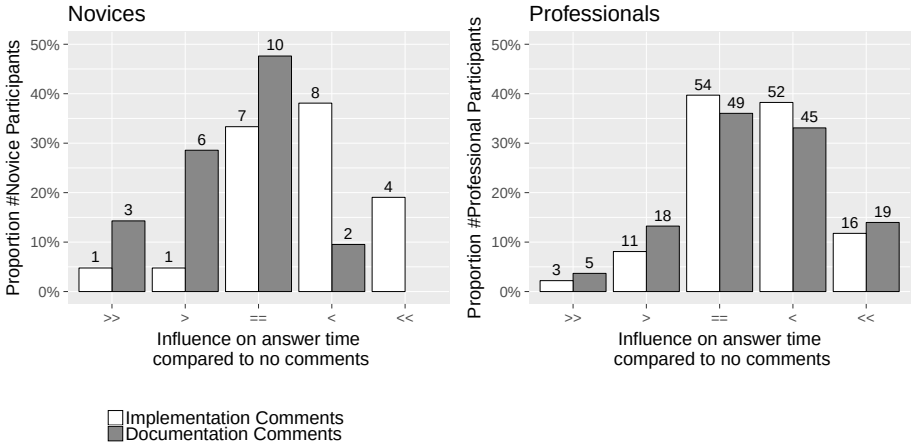


Fig. 5: Subjective assessment on the impact of comments on execution time. (>> high increase, > moderate increase, == no influence, < moderate decrease, << high decrease).

Table 9: Subjective comments of the participants.

Statement	# Mentioned	
	Novices	Professionals
Identifiers are more important than comments	1	13
Mistrusted comments	0	12
Ignored comments	1	6
Comments (would) have been useful	1	6
Use comments (i.e., JavaDoc) for their purpose	0	5
Tests are more important than comments	0	4
Input/output helpful, but of limited value	0	2
Comments rather disruptive than helpful	0	2
Do not repeat yourself	0	2
Examples are useful	0	1
Comments tempt to implement long methods	0	1

In addition, we also received statements that criticized the usage of comments or the design of our tasks. We remark, however, that many issues raised in our experiment are due to us reducing the impact of other variables than comments (e.g., anonymized identifiers). Also, we received positive feedback on the design and tasks and the overall critics indicate that the tasks varied in difficulty.

While the critics we present in Table 9 are hardly representative, they still provide insights into the usage of comments in practice. The main critic we received were the anonymized identifier names. As it is also indicated in other experiments (Takang et al 1996; Anquetil and Lethbridge 1998; Lawrie et al 2007; Hofmeister et al 2017), clean code development and self-explaining identifiers are seen as more important than comments to support software comprehension:

> "According to the clean code motto, comments have to be avoided. After working with legacy code for years, I can only tell to do so."

Despite missing identifiers, developers often mistrust and ignore comments:

Table 10: Results of previous studies compared to our study.

Previous Study	Result	Consistent
Sheppard et al (1978)	No influence of comments	●
Woodfield et al (1981)	Improvements only for modularized code	◖
Norcio (1982)	Improvements only for unindented code	◖
Dunsmore (1985)	Improvements	○
Tenny (1985)	Marginally significant improvements	○
Tenny (1988)	Improvements for monolithic code	◖
Takang et al (1996)	Improvements	○
Nurvitadhi et al (2003)	Improvements	○
Salviulo and Scanniello (2014)	Improvements for novices, but not for professionals	◖
Börstler and Paech (2016)	No influence of comments	●

○: Not consistent, ◖ : Partly consistent, ●: Consistent

> "You have to believe."

Both points are strongly connected to experiences that comments are not updated, of poor quality, or misleading. Several participants prefer to rely on other mechanisms to ensure that the code works as intended, mainly testing and debugging:

> "Don't trust the comment ... only the debugger tells the truth!"

The core message of several responses is the explicit statement that:

> "In doubt, the comment is not helpful, as the code comprises the final truth of what is happening."

Nonetheless, several of our participants also stated that the comments have been useful, or would have been useful. They especially argued that they are necessary, due to the missing identifiers challenging their code comprehension.

Discussion The subjective assessment of our participants indicates that they see a more positive impact of comments than we found. Some participants revealed that they mistrust or deliberately ignore comments and concentrate their focus on the code itself, usually identifiers. Nevertheless, some participants stated that particular comments support program comprehension in certain situations, for example, documentation comments help to understand interfaces. The investigation of these phenomena will be part of future work.

Regarding **RQ$_3$**, the subjective assessments indicate that developers have a more positive anticipation of comments than there may actually be. In contrast, the open responses also reveal several concerns that can guide further research to make comments more valuable and reliable.

4.4 Comparing the Results to Previous Studies

In this section, we finally compare the results of the previously identified studies (cf. Section 2.2). To this end, we summarize the findings again within Table 10 and compare these to our own results. Thus, we provide a direct, explicit comparison, as is recommended by Carver (2010). We remark again that this comparison is mainly concerned with **RQ$_1$** and only for a single study (Salviulo and Scanniello 2014) with **RQ$_2$**.

We can see in Table 10 that we obtain the same results as Sheppard et al (1978) and Börstler and Paech (2016) who find no influence of comments at all, neither in correctness nor time. Moreover, Börstler and Paech (2016) indicate the same effect that we found: Comments seem to be positively associated, while we cannot find any evidence for this.

Other studies find positive impacts of comments for specific situations, for example, on unintended or monolithic code (Norcio 1982; Tenny 1988) – which we cannot derive. Identical to Salviulo and Scanniello (2014), we find no influence of comments on professionals, but are not consistent for novices. All the previous studies discuss (e.g., Börstler and Paech (2016)) and indicate that there are several factors that can hardly be separated, but have an impact on core comprehension. Thus, more extensive studies seem still necessary.

Our results are conflicting with those of four other studies. However, as aforementioned, these studies comprise also other factors that may threaten their validity. The partial and full overlaps to other studies still give us confidence that our results are valid. Considering our results and comparing them to these studies, we argue that especially the qualitative insights may be of high value to understand program comprehension in software engineering: Several factors influence program comprehension and – despite being considered positively – comments may have only a small impact in general. This may be a result of the subjectively identified problems of comments (e.g., consistency, currentness, usage scenario).

5 Threats to Validity

In this section, we discuss threats to the validity of our work. We follow common classifications (Cook and Campbell 1979; Perry et al 2000; Wohlin et al 2012) and consider threats to *construct*, *internal*, and *external* validity.

Construct Validity A potential threat to the construct validity of our experiment are participants misinterpreting our survey tasks and questions. In particular, the online survey did not allow participants to ask for clarification. We addressed this threat by asking the participants questions on their opinions on comments in general to gain more qualitative insights.

Another threat is the type of commented code we provided. Since developers use different styles, length, and languages to comment code, they may not have been ideal for all participants. However, in an industrial setting, developers are also forced to understand code and comments of others. For this reason, we argue that this poses no threat to the results of our work.

Internal Validity A first threat to the internal validity is that our participants got disturbed while answering the online survey. This may result in longer answering times and distort the measured time. However, we encountered this problem by removing conspicuous submissions (i.e., with too long/short answer times) or those participants who stated they were interrupted. Even though some results could be distorted by shorter interruptions, we argue that the effect on our results is negligible and represents real-world scenarios.

Due to the unsupervised structure of our study, it may be possible that participants participated multiple times in our study. This may distort our results.

However, neither our recorded data nor the checked submissions indicate that any participant did so.

Another threat is the difference in the participants' knowledge. To overcome this, we assessed the programmers' experience automatically and split participants into groups of novices and professionals. Moreover, we mentioned at the beginning of the study that the programming tasks are based on Java so that only programmers familiar with Java would participate. Thus, we were able to distinguish and compare the influence of their experience.

Finally, participants could have become familiar with the tasks and learned while they were solving tasks. Thus, the time to find the solution may be lower for later tasks and bias our findings. Even though we provided the same order of the tasks, we think that due to the varying task types – applying, bug-fixing, and extending – and differences in the tasks, there exists no learning effect. Moreover, our results do not indicate such an effect.

External Validity One threat to the external validity is the type of code examples. Because we did not use real-world examples, due to their length and their non-sufficient implementation and JavaDoc comments at the same time, our findings may not be completely transferable into practice. In particular, the results may look quite different if we apply the tasks on a large scale code basis or if we analyze a larger set of tasks. In the latter case, we may also derive more statistically significant results when considering the size or the particular content of comments. To address this threat, we performed an additional qualitative study to discuss and verify the results.

In our experiments we target only German developers, and, thus, provided comments and tasks in German as well. This may restrict our results just on this particular set of developers. However, using the native language to describe something is more suitable to improve comprehension and avoid misunderstandings. Thus, we argue that this does not threaten our results. To enable repetitions of this experiment and check this threat, we added the translated tasks and comments in the appendix of this article.

Another threat is that we used only code examples in Java. For this reason, some participants may be better based on their experience in this programming language. Also, we think it is questionable if the results will be similar for other programming languages (cf. Section 2.2). Those may not support the same paradigms, comments, or are harder to understand. Still, the findings of this study provide reliable insights into the influence of comments.

6 Conclusions

In this article, we explored the impact of comments on program comprehension. To this end, we conducted an online study, in which participants were asked to perform different and small maintenance tasks with varying types of commented code and to give their subjective assessment on comments. We applied three different types of commented code: implementation comments describing the purpose of code lines, documentation comments describing the purpose of a class or a method, and non-commented code. In total, 277 participants (50 novices and 227 professionals) participated in our study. Based on that, we analyzed three research questions.

With **RQ$_1$**, we analyzed if there exists a difference in the performance when programmers are confronted with differently commented code. We analyzed the correctness of all and the answer times of correct submissions. Our results revealed that differences in both, correctness and answer times are more sporadic than regular. In particular, for professionals, the effect of comments on program comprehension seems to be negligible. Even though existing studies found improvements for novice programmers, we were not able to replicate this consistently.

In **RQ$_2$**, we considered differences between novices and professionals, when confronted with the same type of commented code. We found that, if confronted with documentation comments, in 3 out of 9 tasks professionals achieved a significantly higher proportion of correct answers than novices. This indicates that professionals and novices behave differently when dealing with documentation comments. These findings are coherent with those by Salviulo and Scanniello (2014) and pose the question, whether studies comprising only novice programmers are representative for practice.

Finally, in **RQ$_3$** we asked the participants to self-assess comments and compared their responses to our findings. Most participants consider comments as slightly beneficial for programming, even though our measured results do not replicate this. This indicates that comments may be considered more important for program comprehension than they really are. Overall, the key message reported by some professionals is: **"Only the code explains the code."** Consequently, other factors, such as code complexity, identifiers, and tooling, seem more important than comments.

All in all, the provoking question remains, *whether comments are useful at all?* Despite the negligible effect in our findings, we think they are – but we should not ask for too much. Firstly, as was stated by participants, some comment types may be only beneficial in certain situations, for example, documentation comments for interface descriptions. Secondly, our code samples were small (<30 SLOC) and algorithmic, wherefore they may require less explanation than more complex code samples. We encourage to study whether comments may be more supportive if the code reaches a fairly large size or a certain level of complexity. Finally, comments can serve for further purposes, such as communication among developers (Ying et al 2005) or to enable traceability (Ali et al 2015; Ji et al 2015). The importance of comments in such scenarios is interesting and helpful future work. If the commented code is correlated with higher quality, will be part of the further research. Moreover, we considered the type of the commented code as a source of differences. However, we suggest for further research to investigate varying features of comments, for example, length or quality. The assessment of "quality" of comments is in our opinion very complex and subjective to developers and the particular tasks. Finding criteria of "high quality" comments remains an open question for future work. Regarding this, our observations on certain negative effects of comments, for example, the `@see` annotation on novices' performance, can be the origin for future studies investigating the effects of certain comment elements, such as annotations versus natural language.

Acknowledgments

This research is supported by DFG grant LE 3382/2-1.

References

Ali N, Sharafi Z, Guéhéneuc YG, Antoniol G (2015) An Empirical Study on the Importance of Source Code Entities for Requirements Traceability. Empirical Software Engineering (EMSE) 20(2):442–478

Anquetil N, Lethbridge T (1998) Assessing the Relevance of Identifier Names in a Legacy Software System. In: Proceedings of the 8th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), IBM, pp 213–222

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering Traceability Links Between Code and Documentation. IEEE Transactions on Software Engineering (TSE) 28(10):970–983

Basili VR, Shull F, Lanubile F (1999) Building Knowledge Through Families of Experiments. IEEE Transactions on Software Engineering (TSE) 25(4):456–473

Beck F, Moseler O, Diehl S, Rey GD (2013) In Situ Understanding of Performance Bottlenecks through Visually Augmented Code. In: Proceedings of the 21st International Conference on Program Comprehension (ICPC), IEEE, pp 63–72

Bezerra RMM, da Silva FQB, Santana AM, Magalhaes CVC, Santos RES (2015) Replication of Empirical Studies in Software Engineering: An Update of a Systematic Mapping Study. In: Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, pp 1–4

Boehm BW (1981) Software EngineeringEconomics. Prentice-Hall

Börstler J, Paech B (2016) The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment. IEEE Transactions on Software Engineering (TSE) 42(9):886–898

Briand L, Bunse C, Daly J, Differding C (1997) An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. In: Proceedings of the 5th International Conference on Software Maintenance (ICSM), IEEE, pp 130–138

Buse RP, Weimer WR (2010) Learning a Metric for Code Readability. IEEE Transactions on Software Engineering (TSE) 36(4):546–558

Carver JC (2010) Towards Reporting Guidelines for Experimental Replications: A Proposal. In: Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering (RESER)

Chikofsky EJ, Cross JH (1990) Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software 7(1):13–17

Cook TD, Campbell DT (1979) Quasi-Experimentation: Design & Analysis Issues for Field Settings. Houghton Mifflin

Corazza A, Maggio V, Scanniello G (2015) On the Coherence Between Comments and Implementations in Source Code. In: Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, pp 76–83

Cornelissen B, Holten D, Zaidman A, Moonen L, van Wijk JJ, Van Deursen A (2007) Understanding Execution Traces using Massive Sequence and Circular Bundle Views. In: Proceedings of the 15th International Conference on Program Comprehension (ICPC), IEEE, pp 49–58

Dinno A (2015) Nonparametric Pairwise Multiple Comparisons in Independent Groups using Dunn's Test. Stata Journal 15(1):292–300

Dunsmore HE (1985) The Effect of Comments, Mnemonic Names, and Modularity: Some University Experiment Results. In: Empirical Foundations of Information and Software Science, Springer, pp 189–196

Elshoff JL, Marcotty M (1982) Improving Computer Program Readability to Aid Modification. Communications of the ACM 25(8):512–521

Feigenspan J, Kästner C, Liebig J, Apel S, Hanenberg S (2012) Measuring Programming Experience. In: Proceedings of the 20th International Conference on Program Comprehension (ICPC), IEEE, pp 73–82

Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G (2013) Do Background Colors Improve Program Comprehension in the# ifdef Hell? Empirical Software Engineering (EMSE) 18(4):699–745

Fisher RA (1936) Statistical Methods for Research Workers, 6th edn. Oliver and Boyd Edinbrug, Tweeddale Court London: 33 Paternoste R Row, E.C.

Fluri B, Wursch M, Gall HC (2007) Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes. In: Proceedings of the 14th Working Con-

ference on Reverse Engineering (WCRE), IEEE, pp 70–79
Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley
Gosling SD, Vazire S, Srivastava S, John OP (2004) Should we Trust Web-Based Studies? A Comparative Analysis of Six Preconceptions about Internet Questionnaires. American Psychologist 59(2):93
Hanenberg S, Kleinschmager S, Robbes R, Tanter É, Stefik A (2014) An Empirical Study on the Impact of Static Typing on Software Maintainability. Empirical Software Engineering (EMSE) 19(5):1335–1382
Hofmeister J, Siegmund J, Holt DV (2017) Shorter Identifier Names Take Longer to Comprehend. In: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 217–227
Höst M, Regnell B, Wohlin C (2000) Using Students as Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. Empirical Software Engineering (EMSE) 5(3):201–214
Jalali S, Wohlin C (2012) Systematic Literature Studies: Database Searches vs. Backward Snowballing. In: Proceedings of the 6th International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM, pp 29–38
Jbara A, Feitelson DG (2015) How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking. In: Proceedings of the 23rd International Conference on Program Comprehension (ICPC), IEEE, pp 244–254
Ji W, Berger T, Antkiewicz M, Czarnecki K (2015) Maintaining Feature Traceability with Embedded Annotations. In: Proceedings of the 19th International Software Product Line Conference (SPLC), ACM, pp 61–70
Jiang ZM, Hassan AE (2006) Examining the Evolution of Code Comments in PostgreSQL. In: Proceedings of the 3rd Working Conference on Mining Software Repositories (MSR), ACM, pp 179–180
Juristo N, Vegas S (2009) Using Differences Among Replications of Software Engineering Experiments to Gain Knowledge. In: Proceedings of the 24th International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, pp 356–366
Khamis N, Witte R, Rilling J (2010) Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In: Proceedings of the 9th International Conference on Natural Language Processing and Information System (NLDB), Springer, pp 68–79
Knuth DE (1984) Literate Programming. The Computer Journal 27(2):97–111
Kobayashi K, Kamimura M, Yano K, Kato K, Matsuo A (2013) SArF Map: Visualizing Software Architecture from Feature and Layer Viewpoints. In: Proceedings of the 21st International Conference on Program Comprehension (ICPC), IEEE, pp 43–52
Koenemann J, Robertson SP (1991) Expert Problem Solving Strategies for Program Comprehension. In: Proceedings of the 9th Conference on Human Factors in Computing Systems (CHI), ACM, pp 125–130
Kosar T, Mernik M, Carver JC (2012) Program Comprehension of Domain-Specific and General-Purpose Languages: Comparison using a Family of Experiments. Empirical Software Engineering (EMSE) 17(3):276–304
Kramer D (1999) API Documentation from Source Code Comments: A Case Study of Javadoc. In: Proceedings of the 17th Annual International Conference on Computer Documentation (SIGDOC), ACM, pp 147–153
Krüger J, Gu W, Shen H, Mukelabai M, Hebig R, Berger T (2018) Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In: Proceedings of the 12th Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), ACM, pp 105–112
Krüger J, Wiemann J, Fenske W, Saake G, Leich T (2018) Do You Remember This Source Code? In: Proceedings of the 40th International Conference on Software Engineering (ICSE), ACM, pp 764–775
Kruskal WH, Wallis WA (1952) Use of Ranks in One-Criterion Variance Analysis. Journal of the American Statistical Association 47(260):583–621
Lawrie D, Morrell C, Feild H, Binkley D (2007) Effective Identifier Names for Comprehension and Memory. Innovations in Systems and Software Engineering 3(4):303–318
Mäder P, Egyed A (2015) Do Developers Benefit from Requirements Traceability When Evolving and Maintaining a Software System? Empirical Software Engineering (EMSE) 20(2):413–441

Martin RC (2009) Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education

Martinez M, Monperrus M (2015) Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. Empirical Software Engineering (EMSE) 20(1):176–205

von Mayrhauser A, Vans AM (1995) Program Comprehension During Software Maintenance and Evolution. IEEE Computer 28(8):44–55

McBurney PW, McMillan C (2014) Automatic Documentation Generation via Source Code Summarization of Method Context. In: Proceedings of the 22nd International Conference on Program Comprehension (ICPC), ACM, pp 279–290

McBurney PW, McMillan C (2016) An Empirical Study of the Textual Similarity between Source Code and Source Code Summaries. Empirical Software Engineering (EMSE) 21(1):17–42

Norcio AF (1982) Indentation, Documentation and Programmer Comprehension. In: Proceedings of the 1st Conference on Human Factors in Computing Systems (CHI), ACM, pp 118–120

Nurvitadhi E, Leung WW, Cook C (2003) Do Class Comments Aid Java Program Understanding? In: Proceedings of the 33rd Annual Frontiers in Education, IEEE, vol 1, pp T3C–T3C

Perry DE, Porter AA, Votta LG (2000) Empirical Studies of Software Engineering: A Roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ACM, pp 345–355

Rahman MM, Roy CK, Keivanloo I (2015) Recommending insightful comments for source code using crowdsourced knowledge. In: Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, pp 81–90

Ratol IK, Robillard MP (2017) Detecting Fragile Comments. In: Proceedings of the 32nd International Conference on Automated Software Engineering (ASE), IEEE Press, pp 112–122

Runeson P (2003) Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data. In: Proceedings of the 7th International Conference on Evaluation and Assessment in Software Engineering (EASE), Lund University, pp 95–102

Salviulo F, Scanniello G (2014) Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance: Results from an Ethnographically-Informed Study with Students and Professionals. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, p 48

Schröter I, Krüger J, Siegmund J, Leich T (2017) Comprehending Studies on Program Comprehension. In: Proceedings of the 25th International Conference on Program Comprehension (ICPC), IEEE, pp 308–311

Seiler M, Paech B (2017) Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems. In: Requirements Engineering: Foundation for Software Quality, Springer, pp 174–180

Shakeel Y, Krüger J, von Nostitz-Wallwitz I, Lausberger C, Campero Durand G, Saake G, Leich T (2018) (Automated) Literature Analysis - Threats and Experiences. In: Proceedings of the International Workshop on Software Engineering for Science (SE4Science), ACM, pp 20–27

Sharon D (1996) Meeting the Challenge of Software Maintenance. IEEE Software 13(1):122–125

Sheppard S, Borst M, Curtis B, Love L (1978) Predicting Programmers' Ability to Modify Software. Tech. Rep. TR—7a—3B8100 3, General Electric

Siegel S (1956) Nonparametric Statistics For The Behavioral Sciences. McGraw-Hill Kogakusha, LTD, Tokoyo, Japan

Siegmund J (2016) Program Comprehension: Past, Present, and Future. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 13–20

Siegmund J, Siegmund N, Apel S (2015) Views on Internal and External Validity in Empirical Software Engineering. In: Proceedings of the 37th International Conference on Software Engineering (ICSE), IEEE, vol 1, pp 9–19

Sommerlad P, Zgraggen G, Corbat T, Felber L (2008) Retaining Comments When Refactoring Code. In: Proceedings of the 23rd Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), ACM, pp 653–662

Sridhara G (2016) Automatically Detecting the Up-To-Date Status of ToDo Comments in Java Programs. In: Proceedings of the 9th India Software Engineering Conference (ISEC), ACM, pp 16–25

Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards Automatically Generating Summary Comments for Java Methods. In: Proceedings of the 25th International Conference on Automated Software Engineering (ASE), ACM, pp 43–52

Standish TA (1984) An Essay on Software Reuse. IEEE Transactions on Software Engineering (TSE) (5):494–497

Steidl D, Hummel B, Juergens E (2013) Quality Analysis of Source Code Comments. In: Proceedings of the 21st International Conference on Program Comprehension (ICPC), IEEE, pp 83–92

Storey MAD (2005) Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In: Proceedings of the 13th International Workshop on Program Comprehension (IWPC), IEEE, pp 181–191

Storey MAD, Wong K, Muller HH (1997) How do Program Understanding Tools Affect how Programmers Understand Programs? In: Proceedings of the 4th Working Conference on Reverse Engineering (WCRE), IEEE, pp 12–21

Svahnberg M, Aurum A, Wohlin C (2008) Using Students As Subjects - an Empirical Evaluation. In: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM, pp 288–290

Takang AA, Grubb PA, Macredie RD (1996) The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation. Journal of Programming Languages (JPL) 4(3):143–167

Tan SH, Marinov D, Tan L, Leavens GT (2012) @tComment: Testing JavaDoc Comments to Detect Comment-Code Inconsistencies. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST), IEEE, pp 260–269

Tenny T (1985) Procedures and Comments vs. The Banker's Algorithm. ACM SIGCSE Bulletin 17(3):44–53

Tenny T (1988) Program Readability: Procedures versus Comments. IEEE Transactions on Software Engineering (TSE) 14(9):1271–1279

Tiarks R (2011) What Maintenance Programmers Really Do: An Observational Study. In: Proceedings of the 13th Workshop on Software Reengineering, pp 36–37

Trochim WM, Donnelly JP, Arora K (2016) Research Methods The Essential Knowledge Base, 2nd edn. Cengage Learning, 20 Channel Center Street, Boston, MA 02210, USA

Trumper J, Dollner J, Telea A (2013) Multiscale Visual Comparison of Execution Traces. In: Proceedings of the 21st International Conference on Program Comprehension (ICPC), IEEE, pp 53–62

Vermeulen A (2000) The Elements of Java (TM) Style. Cambridge University Press

Wohlin C (2014) Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM, pp 1–10

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering. Springer

Wong E, Yang J, Tan L (2013) Autocomment: Mining Question and Answer Sites for Automatic Comment Generation. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE), IEEE, pp 562–567

Woodfield SN, Dunsmore HE, Shen VY (1981) The Effect of Modularization and Comments on Program Comprehension. In: Proceedings of the 5th International Conference on Software Engineering (ICSE), IEEE, pp 215–223

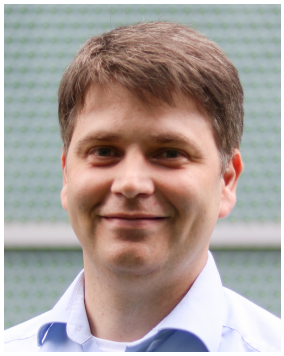
Ying AT, Wright JL, Abrams S (2005) Source code that talks: An Exploration of Eclipse Task Comments and their Implication to Repository Mining. ACM SIGSOFT Software Engineering Notes (SEN) 30(4):1–5

Sebastian Nielebock is a PhD student at the Otto-von-Guericke University of Magdeburg. He received his B.Sc. and M.Sc. degrees in Computer Systems in Engineering from there as well. Since October 2013 he is a member of the Chair of Software Engineering. His research focuses on empirical and automated software engineering, i.e., programming language analysis, automated error detection and automated program repair.

Dariusz Krolikowski is a software developer at IBM Client Innovation Center Germany GmbH in Magdeburg. He received his B.Sc. and M.Sc. degrees in Computer Science at the Otto-von-Guericke University of Magdeburg.

Jacob Krüger is a PhD student and associated researcher at the Databases and

Software Engineering group of the Otto-von-Guericke University of Magdeburg. He received his M.Sc. degree in Business Informatics at the University of Magdeburg in February 2016, has been working as research associate at the Harz University of Applied Sciences Wernigerode, and visited Chalmers University of Technology | University of Gothenburg in Sweden. His research focuses on software-product-line engineering, with particular interests on reverse engineering, code comprehension, costs, and human factors.



Thomas Leich is Professor for Requirements Engineering at Harz University of Applied Sciences in Wernigerode, Germany. He is also Executive Director of METOP GmbH, an affiliate institute to the University of Magdeburg. Since 2001 he worked for several DAX 30 companies as consultant and software architect. In 2004, he initiated FeatureIDE as a part of the FeatureC++ project at the University of Magdeburg. Until today he is responsible for industrial extensions and consulting of FeatureIDE.

Frank Ortmeier is a full professor and head of the "Chair of Software Engineering (CSE)" at the Otto-von-Guericke University of Magdeburg, Germany. He received his Ph. D. degree from the University of Augsburg in 2005. After three years employed as a Post-Doc in Augsburg, he became an associate professor for "Computer Systems in Engineering" in Magdeburg in 2008. Since 2013 he is holding the Chair fo Software Engineering at OvGU. Currently, he is leading several research projects, coordinating the Bachelor' degree program "Computer Systems in Engineering" as well as the Master's degree program "Digital Engineering". He

is a founding member of the university's Center for Digital Engineering, Management and Operations (CeDEMO). His research is driven by the idea of improving engineering tasks with methods from computer science – with a special focus on methods from Software Engineering, formal specification techniques, mobile assistance, and robotics.

A Appendix

In the following, we present our 9 tasks and their solutions. To indicate the comment type, we always use single-line marks (`//`) for implementation comments and multi-line marks (`/** */`) for documentation comments. Versions with no comments contained none of the lines marked this way. The solutions were used as exemplary sketches, but we checked each solution individually.

A.1 Apply Code (Tasks 1-3)

Task 1: Call method `foo()` in such a way that it returns `7`.

```
public class Use1 {
 /**
 * Sums up all values of the given Strings where
 * the Strings are defined by the following numbers:
 * "i" -> 1
 * "v" -> 5
 * "x" -> 10
 *
 * @param strings a String array
 * @return sum of all String values
 */
 public int calculateNumber(String[] strings) {

  int number = 0; // sum of all String values

  for (int i = 0; i < strings.length; i++) {
   if (strings[i].equals("i")) {
    number = number + 1;     // "i" = 1
   } else if (strings[i].equals("v")) {
    number = number + 5;     // "v" = 5
   } else if (strings[i].equals("x")) {
    number = number + 10;    // "x" = 10
   }
  }
  return number;
 }
}
```

Listing 2: Task 1

```
public static void main(String[] args) {

 int number = new Use1().calculateNumber(new String[]{"v", "i", "i"});
 System.out.println(number);

 String numbers = "vii";
 System.out.println(new Use1().calculateNumber(numbers.split("")));
}
```

Listing 3: Task 1 Solution

Task 2: Call method `foo()` in such a way that it returns `"doremi"`.

```
public class Use2 {

 /**
 * Returns the suffix starting at index <code>number</code>. If
 * <code>bool</code> is true, the omitted prefix is appended at the end.
```

```
 *
 * @param string1 a String which must not be null
 * @param number starting index
 * @param bool true if the omitted prefix should be appended at the end
 * @return suffix; followed by the prefix if <code>bool</code> is true
 */
 private String foo(String string1, int number, boolean bool) {

  String string2 = "";

  for (int i = number; i < string1.length(); i++) {
   // suffix starting at number
   string2 = string2 + string1.charAt(i);
  }

  if (bool) {
   // if bool is true, then append the omitted prefix to the string
   for (int j = 0; j < number; j++) {
    string2 = string2 + string1.charAt(j);
   }
  }
  return string2;
 }
}
```

Listing 4: Task 2

```
public static void main(String[] args) {
  System.out.println(new Use2().foo("midore", 2, true));
  System.out.println(new Use2().foo("doremi", 0, false));
}
```

Listing 5: Task 2 Solution

Task 3: Change (only) the list `objectList` in the method `bar` so that the call of this method returns `"Amy"`.

```
public void bar() {

 List<Class1> objectList = new ArrayList<>();

 objectList.add(new Class1("Steve", 25));
 objectList.add(new Class1("John", 42));
 objectList.add(new Class1("Claudia", 19));

 Class1 object = foo(objectList);

 System.out.println(object.string);
}
```

```
public class Use3 {

 /**
 * Contains the name and the age of a person
 */
 class Class1 {

  /** name */
  String string; // name

  /** age */
  int number; // age
```

```
13
14   public Class1(String string, int number) {
15    this.string = string;
16    this.number = number;
17   }
18  }
19
20  /**
21  * Returns the oldest person
22  *
23  * @param objectList list of persons; must contain at least one person
24  * @return the oldest person
25  */
26  public Class1 foo(List<Class1> objectList) {
27   Class1 object = objectList.get(0);
28
29   for (int i = 1; i < objectList.size(); i++) {
30    // compare the age and store the oldest person
31    if (objectList.get(i).number > object.number) {
32     object = objectList.get(i);
33    }
34   }
35   return object;
36  }
37 }
```

Listing 6: Task 3

```
1 objectList.add(new Class1("Amy", 50));
```

Listing 7: Task 3 Solution

A.2 Bug Fixing (Tasks 4-6)

Task 4: The `foo()` method throws a runtime exception for the following input. Fix the error so that the expected result [3, 8] is returned.

```
foo(new int[]{1, 3, 4, 5, 8, 11, 13},
    new int[]{2, 3, 5, 7, 8, 9});
>> [3,8]
```

```
1  /**
2   * Returns a list of numbers that are equal at the same position in 2
        arrays
3   *
4   * @param numbers1 first array
5   * @param numbers2 second Array
6   * @return list of equal numbers
7   */
8  public List<Integer> foo(int[] numbers1, int[] numbers2) {
9   List<Integer> numberList = new ArrayList<>(); // result list
10  int max = numbers1.length; // number of elements to be iterated
11
12  for (int i = 0; i < max; i++) {
13   if (numbers1[i] == numbers2[i]) {
14    // if the i-th values of both arrays are equal, add it to result list
15    numberList.add(numbers1[i]);
16   }
17  }
18  return numberList;
19 }
```

Listing 8: Task 4

```
int max = Math.min(array1.length, array2.length);
```

Listing 9: Task 4 Solution

Task 5: The `foo()` method contains an error. Fix it so that the expected results are returned:

```
foo("abcd", "acbd")
  >> 1
foo("abcd", "badc")
  >> 2
foo("abcdef", "defabc")
  >> 3
```

It can be assumed that both strings have the same length and are not `null`.

```
/**
 * Returns the number of transpositions between two strings.
 *
 * One transposition is defined as a single switch of two characters.
 *
 * @param string1 first string,  not null
 * @param string2 second string, not null
 * @return number of transpositions between two strings.
 */
public int foo(final String string1, final String string2) {
 int number = 0; // store the number of different characters
 for (int i = 0; i < string1.length(); i++) {
  if (string1.charAt(i) != string2.charAt(i)) {
  // increment number, if two characters are not equal
   number++;
  }
 }
 return number; // return the number of transpositions
}
```

Listing 10: Task 5

```
return number / 2;
```

Listing 11: Task 5 Solution

After the study was completed, we noticed that our solution handles only strings with an even number of characters, for example, for `"abc"` and `"cab"` the code would calculate only one transposition due to integer division. However, all participants seemed to be unaware of this problem as none of the given answers handles this scenario. Thus, we accepted all answers that solve the task for strings with an even number of characters.

Please notice, that this task does not use the Hamming distance, as a transposition is defined as a single switch of two characters. Thus, we considered `"abcd"` and `"efgh"` as invalid input, because no chars within the string are switched.

Task 6: Fix the compile-time error in `foo()`.

```
/**
 * Returns the suffix starting at the first occurrence of the character
      ch.
 *
 * @param string string
 * @param ch character to start the suffix with
 * @return suffix starting at the first occurrence of the character
 *          empty string, if the string does not contain the character
 */
```

```
9  public String foo(String string, char ch) {
10  for (int i = 0; i < string.length(); i++) {
11   if (string.charAt(i) == ch) {
12    // return suffix starting at the first occurence of the character ch
13    return string.substring(i + 1);
14   }
15  }
16 }
```

Listing 12: Task 6

```
1   return "";
```

Listing 13: Task 6 Solution

A.3 Extend Code (Tasks 7-9)

Task 7: Extend the method `foo` with an `int` parameter, which is returned if `number` equals `0`. Example:

```
foo(new int[]{5,13,31}, 7);
  >> 7
```

```
1  public class Extend1 {
2
3  /**
4   * Adds up all even numbers in an array.
5   *
6   * @param numbers int-array
7   * @return return Sum of all even numbers in the array
8   */
9   public static int foo(int[] numbers) {
10   int number = 0; // save intermediate results
11
12   for (int i = 0; i < numbers.length; i++) {
13    if (numbers[i] % 2 == 0) {
14     // the i-th number is added to the result if it can be divided by 2
15     number = number + numbers[i];
16    }
17   }
18   return number;
19  }
20
21  public int foo2(int[] numbers, int other) {
22   int number = 0;
23
24   for (int i = 0; i < numbers.length; i++) {
25    if (numbers[i] % 2 == 0) {
26     number = number + numbers[i];
27    }
28      }
29     return number == 0 ? other : number;
30  }
31
32  public void main() {
33   System.out.println(foo(new int[]{1, -2, 3, 4}));
34   System.out.println(foo(new int[]{1, 5}));
35  }
36
37  public static void main(String[] args) {
38   new Extend1().main();
39  }
40 }
```

Listing 14: Task 7

```
public static int foo(int[] numbers, int fallback) {
 int result = 0;

 for (int i = 0; i < numbers.length; i++) {
  if (numbers[i] % 2 == 0) {
   result = result + numbers[i];
  }
  if (result == 0)
   return fallback;
  return result;
 }

 public void main() {
  System.out.println(foo_solution(new int[]{1, -2, 3, 4}, 99));
  System.out.println(foo_solution(new int[]{1, 5}, 99));
 }
}
```

Listing 15: Task 7 Solution

Task 8: Extend method `foo()` to ignore `null` Strings for the output.

```
String[] input = {"Tic", null, "Tac", "Toe"};
join(input, ",");

>> Tic,Tac,Toe
```

```
public class Extend2 {

 /**
  * Concatenates all entries of a String-array to a single String,
  * separeted by the defined symbol.
  *
  * @param strings a String-array
  * @param ch separating smybol
  * @return concatenation of Array-entries
  * @see java.lang.StringBuilder
  */
 public static String foo(final String[] strings, char ch) {
  // Resulting String is build with a StringBuilder
  final StringBuilder builder = new StringBuilder();
   for (int i = 0; i < strings.length; i++) {
    if (i > 0) {
     // the separating smybol is included before each String - excpet
          for the first one
     builder.append(ch);
    }
   builder.append(strings[i]); // i-th String is added to the output
   }
  return builder.toString();
 }

 public static void main(String[] args)
  String output1 = foo(new String[]{"Tic", "Tac", "Toe"}, ',');

  System.out.println(output1);

  String[] input2 = {"Tic", null, "Tac", "Toe"};
  String output2 = foo(input2, ',');
  System.out.println(output2);
  String output3 = foo(input2, ',');
```

```
34    System.out.println(output3);
35   }
36  }
```

Listing 16: Task 8

```
1  public static String foo(final String[] strings, char separator) {
2   final StringBuilder builder = new StringBuilder();
3   for (int i = 0; i < strings.length; i++) {
4    if (strings[i] == null)
5     continue;
6     if (i > 0) {
7      builder.append(separator);
8     }
9    builder.append(strings[i]);
10  }
11  return builder.toString();
12 }
```

Listing 17: Task 8 Solution

Task 9: Extend `Class2` with a method `bar()` of the return type `Integer` that reverses the operation of `foo()`.

```
1  public class Extend3 {
2
3   /**
4   * data structure to store a history of numbers
5   */
6   class Class2 {
7    // save numbers in a list
8    LinkedList<Integer> numberList = new LinkedList<>();
9    // pointer to current position in the list
10   int number1 = 0;
11
12   /**
13   * Insert a number at the current position to the history,
14   * potentially following numbers are removed.
15   *
16   * @param number2 number to that shall be added
17   */
18   public void qux(Integer number2) {
19    while (number1 < numberList.size()) {
20     // all numbers after the pointer are removed
21     numberList.removeLast();
22    }
23    // the new number is added at the end and the pointer increased
24    numberList.add(number2);
25    number1++;
26   }
27
28   /**
29   * Removes the last number in the history and returns the previous
         number.
30   * If there is no number in the history, return null.
31   *
32   * @return the previous number of the history
33   */
34   public Integer foo() {
35    if (number1 > 0) {
36     // pointer is not at the beginning of the list - decrease pointer by
37     // 1 and return current value
38     number1--;
39     return numberList.get(number1);
40    } else {
```

```
41      return null; // pointer is at the beginning - return null
42     }
43    }
44   }
45 }
```

Listing 18: Task 9

```
1  public Integer barMy() {
2   if (number1 < numberList.size()) {
3    number1++;
4    return numberList.get(number1);
5   } else {
6    return null;
7   }
8  }
9
10 public Integer bar() {
11  if (number1 <= 0) {
12   number1++;
13   return numberList.get(number1);
14  } else {
15   return null;
16   }
17  }
18 }
```

Listing 19: Task 9 Solution