`#ifdef` Directives and Program Comprehension: The Dilemma between Correctness and Preference

Wolfram Fenske
pure-systems GmbH, Magdeburg, Germany
Email: wolfram.fenske@pure-systems.com

Jacob Krüger, Maria Kanyshkova, Sandro Schulze
Otto-von-Guericke University Magdeburg, Magdeburg, Germany
Email: {jkrueger | sanschul}@ovgu.de

***Abstract*—Many organizations and open-source projects use the C preprocessor (CPP) to implement configurability in their software systems. Despite extensive research, existing studies on the effects of CPP use on program comprehension are still limited to experiences, opinions, and empirical studies with narrow scopes. So, it is unclear whether the CPP actually leads to what is sometimes referred to as "`#ifdef` hell." In this paper, we expand the existing evidence on program comprehension in the presence of CPP directives, but we also highlight a surprising dilemma. We conducted an empirical study, including an experiment and a questionnaire, on the impact of refactoring CPP directives with 521 experienced software developers. The results indicate that, in contrast to previous findings, comprehension performance slightly worsened in terms of correctness when our participants worked on code with refactored CPP directives. However, in alignment with previous findings, they preferred the refactored code, considering it more comprehensible and easier to work with. This dilemma of objective performance versus subjective preference is a surprising outcome that has not been found before. We argue that our work motivates the need for more studies to understand this dilemma—which may significantly impact common beliefs in research and practice.**

***Index Terms*—Configurable Systems, Preprocessors, Program Comprehension, Refactoring, Empirical Study**

I. Introduction

The C preprocessor (CPP) is a simple, yet effective text-based tool for implementing configurability in a software system following the *annotate-and-remove* paradigm [2], [11], [20]. Its *conditional compilation* directives (e. g., `#ifdef`, `#endif`) allow developers to write code sections whose presence or absence is controlled by a *macro* (i. e., a configuration option). As an example, consider Line 23 on the left side of Fig. 1, where the macro `USE_LONG_FNAME` makes the next line of code optional. During preprocessing, the CPP evaluates all conditional compilation directives, removing the code between directives whose corresponding macro is undefined. In the example in Fig. 1, Line 24 would be removed if the macro `USE_LONG_FNAME` were undefined during preprocessing. Otherwise, it would be compiled into the system.

The CPP is used in many open-source and industrial software systems from numerous domains, allowing these systems to be tailored to specific customer demands, safety regulations, and resource restrictions [20], [21]. Prominent systems using the CPP include the Linux Kernel, which comprises over 26 million lines of code and over 15 thousand configuration options, the Apache web server, and Hewlett-Packard's printer firmware [11], [20], [21], [39]. Interestingly, while developers use the CPP regularly, it is still heavily criticized for numerous issues that may or may not be problematic. One of these issues that has been prominently reported and that we particularly focus on (cf. Sec. II) is the *undisciplined* use of CPP directives [10], [15], that is, directives that do not align with syntactic units [21]. Moreover, the CPP is suspected to impede program comprehension [3], [5], [19], [22], [24], [25], [27], [29], [34], [38], to foster code scattering and tangling [3], [14], [38], to increase fault proneness [1], [3], [9], [26], [28], [30], and to harm maintainability [4], [7], [25]. However, besides the fact that some of the aforementioned studies exhibit contradicting results, most of them either rely solely on source code and repository analysis or on small-scale experiments whose participants are mostly students. Solely Medeiros et al. [25], [27], Malaquias et al. [24], and Muniz et al. [30] conducted empirical studies involving experienced practitioners.

In this paper, we provide insights into developers' comprehension of the CPP with a study that is the first to investigate how well developers' objective correctness during program comprehension aligns with their subjective preference for a certain style of CPP use. To this end, we conducted a large-scale study with 521 developers from various GitHub projects, such as Linux, FreeBSD, and PostgreSQL. In particular, we designed an online study in which we showed the participants examples of C functions that contained CPP directives for configurability. We manually refactored four of these examples in order to reduce the complexity of the CPP directives, while keeping the functionality and configuration options equivalent (see Sec. III-C for details on the refactorings). Our goal was to investigate whether turning complex, fine-grained directives into simpler, coarse-grained directives would improve program comprehension in a realistic setting. For this purpose, we chose code examples from established open-source projects and performed all refactorings that were applicable. Even though this design prevented us from studying different refactorings individually, it avoided the use of artificial examples. For each example, the participants completed two program comprehension tasks and subsequently rated the example's code quality, especially regarding CPP use. To analyze the impact of refactoring, half of our participants worked on the original code of an example, while the other half worked on the refactored code. Our results provide insights into how developers perceive different styles of using CPP directives and how these perceptions align with their objective comprehension performance. In summary, we make the following contributions in this paper:

- We analyze our participants' correctness while solving two program-comprehension tasks on five different examples. So, we gain insights into whether refactoring CPP directives impacts developers' objective program comprehension.
- We analyze our participants' subjective preference regarding our examples. This enriches our quantitative data with qualitative responses and shows the personal opinions of our participants concerning the use of CPP directives.
- We compare and discuss the implications of our quantitative and qualitative results.
- We provide a replication package comprising our study design and our anonymized data.[1]

The results show that slightly fewer participants solved the programs comprehension tasks correctly on the refactored code than on the original code, which partly contradicts previous findings. At the same time, our participants expressed a clear subjective preference for the refactored code. This surprising dilemma raises a number of questions, for instance: Why are the subjective preferences of developers in line with previous findings, while the objective correctness contradicts them? Are certain patterns of CPP use (e. g., undisciplined directives) really as problematic as assumed? For practitioners and tool developers, our results suggest that coding style guides as well as future analyses and programming tools could benefit from relying more on empirical data and less on subjective preferences or "common sense."

II. Related Work

Several researchers have discussed the CPP as a variability mechanism from a theoretical point of view [4], [5], [14], [38], also considering program comprehension. While such works are based on expert knowledge and sound reasonable, our work is more closely related to empirical studies, which derive their insights from observations. We discuss such studies next, distinguishing between (1) descriptive, (2) measurement, (3) correlational, and (4) experimental studies.

Descriptive studies (1) elicit insights into preprocessor use through interviews [25], case studies [38], and qualitative analyses [1], [3], [7], [18], [30]. *Measurement studies (2)* employ metrics to quantify preprocessor use [3], [7], [16], [20], [21], [23], [26], [31]. For example, they quantify how frequently undisciplined directives occur in open-source software [21]. *Correlational studies (3)* investigate whether measurements of preprocessor use correlate with some property of interest [8], [9], [11], such as fault proneness [9]. The study in this paper belongs to the last category, *experimental studies (4).*

The core idea of an experimental study is to manipulate one aspect of preprocessor use and test whether that manipulation affects an outcome of interest. In our study, we investigate the highly human-centric activity of program comprehension [33], [41]. For this reason, we focus on experiments with human participants, which we summarize in Tbl. I. For each study,

[1]https://doi.org/10.5281/zenodo.3972411

TABLE I: Related experiments with human subjects.

Study	**# Part.**		**Manipulated Aspect**	**Measurements**
	Nov.	**Prof.**		
[6] (1)	43	0	colors	C_{c+t}, M_{c+t}, S
[6] (2)	20	0	colors	S
[6] (3)	14	0	colors	C_{c+t}, M_{c+t}, S
[19]	25	6	colors	C_c, C_t, S
[29]	63	6	# features	M_{c+t}
[30]	0	110	faults	M_c
[24] (1)	0	99	discipline	S
[24] (2)	64	0	discipline	M_{c+t}
[25]	0	202	discipline	S
[27] (1)	0	246	discipline	S
[27] (2)	0	$\leq$28	discipline	S
[34]	19	0	discipline	C_{c+t}, M_{c+t}
This study	0	521	complexity	C_c, S

C_c, C_t: Correctness / time for comprehension tasks;
M_c, M_t: Correctness / time for maintenance tasks;
S: Subjective opinion
Nov.: Novices; Prof.: Professionals

we list the number of participants (novices and professionals), the manipulated aspect (e. g., directive discipline), and the measurements used. For publications comprising multiple experiments, we append a suffix (i. e., 1, 2, 3). Unless the authors reported their own classification, we counted undergraduate, graduate, and PhD students as novices and considered industrial, GitHub, and post-doc developers as professionals. We distinguish between three ways of measuring program comprehension: comprehension tasks (C), such as "How many variants of this code are possible?"; maintenance tasks (M), such as locating bugs and suggesting fixes; and subjective opinions (S), such as "How do you rate the code's readability?" For comprehension and maintenance tasks, the response time (t), correctness (c), or both ($c{+}t$) can be measured.

Four controlled experiments [6], [19] studied background colors as a replacement for textual directives. In the first experiment, background colors increased both program comprehension correctness and speed [19]; in the other three experiments, only speed increased but not correctness [6]. Interestingly, the participants of all four experiments preferred background colors. In a controlled experiment involving students and post-docs, higher degrees of configurability caused the speed and accuracy of bug finding to deteriorate [29]. Later, it was demonstrated that even professional developers have difficulties to identify faults in configurable code [30]. These experiments are complementary to ours, as we are not concerned with background colors or identifying faults.

Six experiments [24], [25], [27], [34], investigated the effect of directive discipline on program comprehension, four relying solely on subjective opinions. Medeiros et al. [25], [27] used online questionnaires to ask GitHub developers to rate the quality of code examples, revealing preferences for disciplined directives. Malaquias et al. [24] and Medeiros et al. [27] identified instances of undisciplined directives in open-source projects, refactored them into disciplined directives and sent pull requests to the respective maintainers. Overall, 99 and 28 maintainers, respectively, were involved. In both experiments, the majority of the pull requests was accepted.

In the remaining two experiments [24], [34], students completed comprehension and maintenance tasks on code with disciplined and undisciplined directives. For the first experiment, directive discipline had no significant effect [34], but in the other one, it was clearly beneficial regarding correctness and response times [24].

So, there is evidence that disciplined directives are *subjectively preferred* by developers [24], [25], [27] and increase the number of *objectively correct* solutions [24]. However, this evidence originates from different experiments, involving different participants working on different code examples. Complementary, we collected both *objective* and *subjective* measures for *the same code examples* in our experiment, allowing us to connect both perspectives without threatening a comparison. Consequently, our study fills the open gap of studying and comparing objective correctness and subjective preference regarding program comprehension in the presence of `#ifdef` directives. Moreover, we involved a larger number of experienced subjects, strengthening the empirical evidence.

III. Methodology

In this section, we report our *research questions*, *code examples*, *refactorings*, *study design*, and *selection of participants*.

A. Research Questions

The goal of our study was to investigate to what extent refactoring CPP directives (e. g., disciplining them) impacts the program comprehension of software developers, combining subjective and objective measurements. We defined two research questions:

RQ$_1$ *How do developers perform during program comprehension when facing* CPP *directives of varying complexity?*
For **RQ$_1$**, we aimed to obtain quantitative results by conducting an experiment. To this end, we measured our participants' correctness while solving program-comprehension tasks.

RQ$_2$ *What are the subjective preferences of developers considering the comprehensibility of the source code?*
For **RQ$_2$**, we aimed to understand whether developers prefer refactored directives over the original ones based on a questionnaire. To this end, we had our participants rate the code quality on Likert-scales. Moreover, they could expand on their ratings with free-text comments.

The *independent variable* in our study are the *code examples* (either original or refactored) that we showed the participants. For **RQ$_1$**, the *dependent variable* is whether our participants solved our program-comprehension tasks *correctly.* For **RQ$_2$**, the *dependent variable* is the *quality rating* assigned by our participants. Additionally, we *control* for differences in development experience. Note that we do not consider response time as a dependent variable since (i) our objective was to measure correctness and (ii) our experimental setup (an online questionnaire) did not allow for reliable time measurements. Still, we did exclude unreasonably fast responses, that is, responses from users who just clicked through the questionnaire. In summary, we obtained quantitative measures on program-comprehension correctness, qualitative insights into developers' preferences, and the possibility to compare both.

B. Code Examples

In our study, our participants were shown five code examples, and for each example they had to perform two program comprehension tasks and rate the usage of CPP directives. As code examples, we used functions from Vim and Emacs, two real-world text editors. The functions stem from a dataset of Fenske et al. [7], which comprises C code with different extents of code smells concerning the use of CPP directives. For our study, *we selected examples with particularly high smelliness values* because we assumed that refactoring them would produce a strong impact on our participants' comprehension performance and preferences.

We picked the examples `vim18`, `vim15`, `vim13`, `emacs12`, and `emacs11`. To control for differences in program comprehension and address our research questions, we refactored four examples (except `vim18`)—aiming to improve the code quality regarding CPP directives. As a result, we used the following code examples in our study (cf. Sec. III-D):

- The baseline (`vim18`) to introduce our study and to compare directly between the two groups of participants (explained in Sec. III-D).
- The original (smelly) code examples: `vim15`, `vim13`, `emacs12`, and `emacs11`.
- The refactored examples: `vim15_R`, `vim13_R`, `emacs12_R`, and `emacs11_R`.

Next, we describe the refactorings we performed (Sec. III-C) and how we used them in our study (Sec. III-D).

C. Refactorings

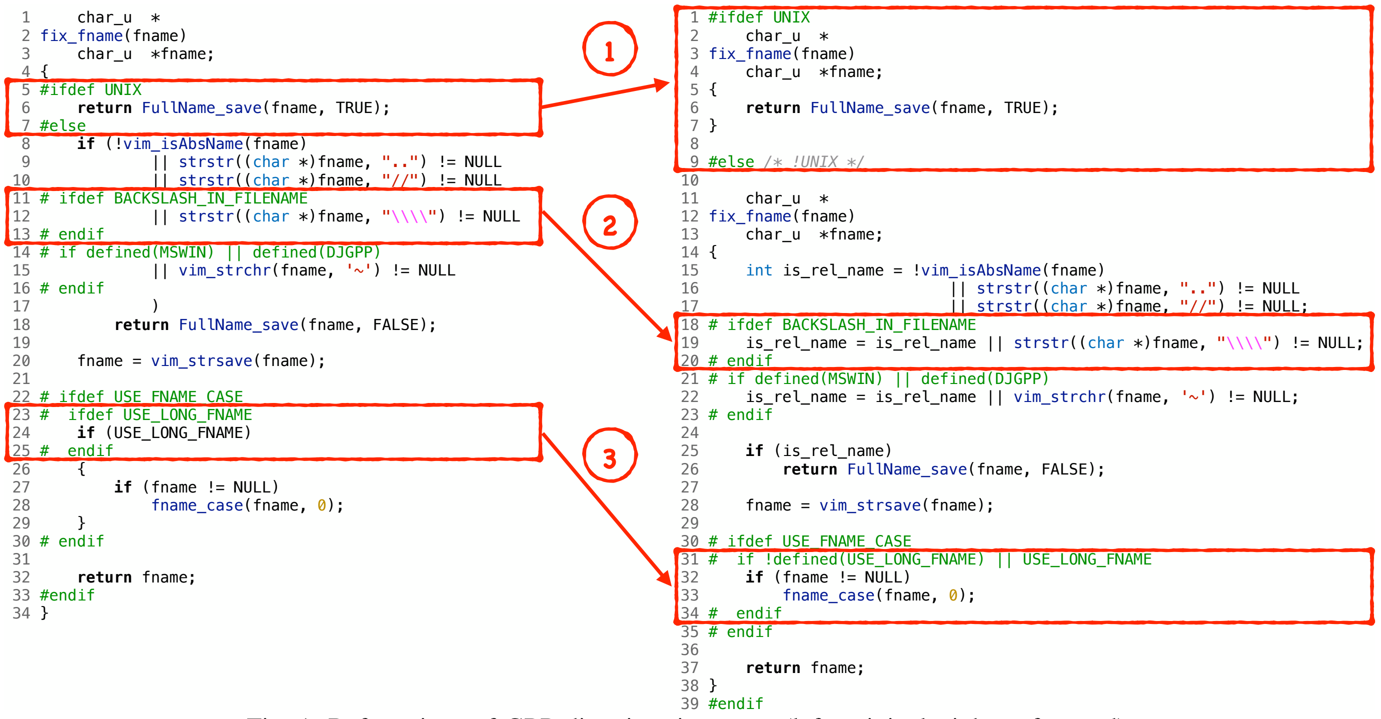
Our objective while refactoring the examples was to simplify the complexity of the CPP directives, while, at the same time, preserving the underlying C code. The refactored code had exactly the same functionality and configuration options as the original code and adhered to the same indentation policies, both for the C code and for nested CPP directives. If comments clarified which condition an `#else` or `#endif` belonged to, we inserted identical comments in the refactored code.

We applied the following three refactorings:

R$_1$ *Extract alternative function* (applied three times)
We refactored one function into two when large blocks of code were enclosed in CPP directives. This was the case for `vim15`, `emacs12`, and `emacs11`.

R$_2$ *Discipline directives* (applied five times)
We refactored one undisciplined directive in `vim15` and four in `vim13`, following the advice of Medeiros et al. [27]. The Emacs examples were free of undisciplined directives.

R$_3$ *Unify compile- and runtime-time variability* (applied once)
In `vim15`, we found a piece of code that mixed compile-time and runtime-time variability, namely an `#ifdef` and an `if` statement that were controlled by the same CPP macro. We refactored this code to comprise only compile-time variability (i. e., `#ifdef` directives). None of our other code examples had to be refactored in this manner.



```
 1     char_u  *
 2 fix_fname(fname)
 3     char_u  *fname;
 4 {
 5 #ifdef UNIX
 6     return FullName_save(fname, TRUE);
 7 #else
 8     if (!vim_isAbsName(fname)
 9             || strstr((char *)fname, "..") != NULL
10             || strstr((char *)fname, "//") != NULL
11 # ifdef BACKSLASH_IN_FILENAME
12             || strstr((char *)fname, "\\\\") != NULL
13 # endif
14 # if defined(MSWIN) || defined(DJGPP)
15             || vim_strchr(fname, '~') != NULL
16 # endif
17             )
18         return FullName_save(fname, FALSE);
19
20     fname = vim_strsave(fname);
21
22 # ifdef USE_FNAME_CASE
23 #  ifdef USE_LONG_FNAME
24     if (USE_LONG_FNAME)
25 #  endif
26     {
27         if (fname != NULL)
28             fname_case(fname, 0);
29     }
30 # endif
31
32     return fname;
33 #endif
34 }
```

```
 1 #ifdef UNIX
 2     char_u  *
 3 fix_fname(fname)
 4     char_u  *fname;
 5 {
 6     return FullName_save(fname, TRUE);
 7 }
 8
 9 #else /* !UNIX */
10
11     char_u  *
12 fix_fname(fname)
13     char_u  *fname;
14 {
15     int is_rel_name = !vim_isAbsName(fname)
16                             || strstr((char *)fname, "..") != NULL
17                             || strstr((char *)fname, "//") != NULL;
18 # ifdef BACKSLASH_IN_FILENAME
19     is_rel_name = is_rel_name || strstr((char *)fname, "\\\\") != NULL;
20 # endif
21 # if defined(MSWIN) || defined(DJGPP)
22     is_rel_name = is_rel_name || vim_strchr(fname, '~') != NULL;
23 # endif
24
25     if (is_rel_name)
26         return FullName_save(fname, FALSE);
27
28     fname = vim_strsave(fname);
29
30 # ifdef USE_FNAME_CASE
31 #  if !defined(USE_LONG_FNAME) || USE_LONG_FNAME
32     if (fname != NULL)
33         fname_case(fname, 0);
34 #  endif
35 # endif
36
37     return fname;
38 }
39 #endif
```

Fig. 1: Refactorings of CPP directives in `vim15` (left: original, right: refactored).

These refactorings were motivated by evidence from the literature [24], [25], [27]. In particular, the participants of the survey of Medeiros et al. [25] expressed that they prefer alternative function definitions over using conditional CPP directives in functions to solve portability issues. Thus, R_1 should improve the code. R_2 was motivated by the frequently documented aversion of developers to undisciplined directives [24], [25], [27]. At least some of our participants share this aversion:

On `vim15`
"This is a prime example of preprocessor abuse. Much too fine grained. Very difficult to reason about. Impossible to test [...]"

We show the code in question on the left side of Fig. 1. Even though the participant does not mention "discipline" by name, their remark goes to the heart of the critique of undisciplined directives. Based on such remarks and building on previous findings, we believe that our refactorings are reasonable and that practitioners would agree that we tackle the right problems. We remark that we applied refactorings due to their applicability on each example, aiming to understand the impact of refactoring CPP directives overall, not the impact of individual refactorings.

We manually applied the refactorings and validated the results with colleagues from other organizations. This way, we discovered a small number of errors, which we fixed before deploying the actual study. Some participants complained about additional errors in the refactored code, but we found that all of these complaints except for one were unjustified. The one remaining error affected the refactored version of `emacs11` (i. e., `emacs11_R`), where we inadvertently changed an `#if` into an `#ifdef`, thus slightly changing the syntax and semantics. However, this change was not the problem; the problem was that one question continued to refer to this directive as an `#if`, not an `#ifdef`, which may have confused our participants. Despite this error, the responses to this question were not particularly unusual, which we interpret as meaning that our participants' correctness was not substantially affected. We therefore argue that this error does not threaten our results.

We exemplify our refactorings based on `vim15` in Fig. 1. In this example, we show a function from VIM that translates filenames into a canonical form for internal use. We show the original code on the left side of Fig. 1 and the refactored code on the right. For each of the refactoring types, we applied one refactoring on this example, which we highlight with the red circles and numbers (using the same numbers as before):

- R_1 The first change (see ① in Fig. 1) constitutes an *extract alternative function* refactoring. As a result, the original function definition was split into one definition for UNIX-style operating systems (see Lines 1–8 on the right) and an alternative definition for other operating systems, such as WINDOWS (see Lines 9–39). Although the refactored code is longer overall, the individual function definitions are shorter, more cohesive, and contain fewer nested CPP directives. Consequently, it should become clearer how the CPP directives influence the behavior of the source code.
- R_2 The second change (see ② in Fig. 1) highlights a *discipline directive* refactoring. In the original code on Line 8, there is an `if` statement with a long conditional expression of which several sub-expressions are controlled by CPP directives (see Lines 11–13 and Lines 14–16 on the left). With the refactoring, we extracted the first part of the condition into a variable and the sub-expressions into statements that modify that variable (see Lines 18–20 and Lines 21–23 on the right). So, the refactored version no longer exhibits

undisciplined CPP directives below statement level.

R_3 The third change (see ③ in Fig. 1) highlights a *unify compile-time and runtime-time variability* refactoring. The original code mixed an `#ifdef` and a runtime `if` to essentially encode a logical implication. We refactored this mix into a more explicit version that solely relies on compile-time variability. As a result, we removed mixed variability mechanisms and improved consistency, which should improve program comprehension.

According to previous findings [24], [25], [27], the refactored code on the right side of Fig. 1 should be easier to comprehend.

D. Study Design

One crucial decision we made was to develop two versions of our study (denoted **S1** and **S2** in the following), which differed in the code examples they contained. Each version started with the same baseline example, `vim18`, in its original form. For the examples 2–5, the studies alternated between original and refactored code examples, with **S1** starting with an original example and **S2** with a refactored example (cf. Tbl. II). This design allowed us to compare different versions of the same code in one experiment, while, at the same time, avoiding learning biases. Moreover, the baseline example allowed us to identify systematic differences between the groups working on **S1** and **S2**. In Tbl. II, we display the code examples and the questions that were part of each study. Essentially, the studies comprised three parts, split across six sections. The order in which we presented Q_1 to Q_7 is the same as in Tbl. II. Moreover, for each example, we defined a separate section, consisting of the corresponding source code listing and the questions Q_8 to Q_{11}, also following the order in Tbl. II.

In the first part (*Background*: Q_1 to Q_7 in Tbl. II), we asked our participants for background information. The answers allowed us to control for differences in our participants' age, sex, and programming experience. We formulated these questions following existing guidelines [35].

The second and third part of our study (*Examples* and *For each example* in Tbl. II) were split across the remaining five sections, with each section comprising one code example. In particular, we asked two comprehension questions (i. e., Q_8 defining Task 1 and Q_9 defining Task 2) about each example. For Q_8, we provided a small number of statements about the code from which the participants had to select the correct one. For Q_9, in turn, we defined several options that aligned with the CPP directives in the example and the participants had to "configure" their selection so that a certain line would be executed. We assumed this question to be more challenging, as it required our participants to understand all the CPP directives in the example. Independently of whether the original or the refactored code was shown, we always asked for the same line, not the same line number. To avoid biasing our participants against CPP use, we designed the questions so that the participants needed to understand the CPP directives to give the correct answer, but we formulated the question texts in a way that did not emphasize this focus.

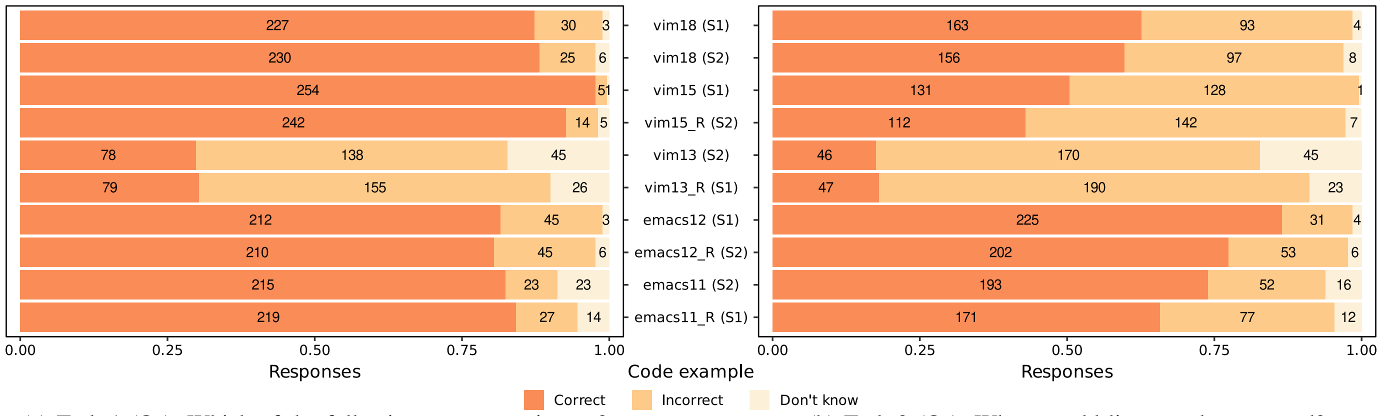
TABLE II: Overview of the questions and possible answers in our study. We mark refactored examples with `_R` and state the corresponding version (S1 or S2). The questions for the examples did not differ, we only adapted the line numbers to refer to the appropriate line of code.

ID	Questions & Answers					
Background						
Q_1	How old are you?					
	○ 15–19 years — ○ 65+ years (5 year periods)					
Q_2	Gender					
	○ Female ○ Male ○ Other ○ Prefer not to tell					
Q_3	How many years of programming experience do you have?					
	Open number					
Q_4	How many years of experience with C/C++ do you have?					
	Open number					
Q_5	Roles in projects					
	Multiple selection (e. g., Developer) and open text					
Q_6	Which open-source projects have you worked on so far?					
	Open text					
Q_7	How would you rank your programming skills in C/C++?					
	○ Beginner ○ Intermediate ○ Advanced ○ Expert					
Examples						
Q_8	Which of the following statements is true?					(*Task 1*)
	Single selection out of					
	`vim18`	S1	`vim18`	S2	5 options	
	`vim15`	S1	`vim15_R`	S2	5 options	
	`vim13_R`	S1	`vim13`	S2	5 options	
	`emacs12`	S1	`emacs12_R`	S2	5 options	
	`emacs11_R`	S1	`emacs11`	S2	6 options	
Q_9	When would line <x> be executed?					(*Task 2*)
	Choosing a combination out of					
	`vim18`	S1	`vim18`	S2	9 conditions	
	`vim15`	S1	`vim15_R`	S2	11 conditions	
	`vim13_R`	S1	`vim13`	S2	11 conditions	
	`emacs12`	S1	`emacs12_R`	S2	7 conditions	
	`emacs11_R`	S1	`emacs11`	S2	9 conditions	
For each example						
Q_{10}	Do you consider the use of preprocessor annotations in the example appropriate?					
	○ Yes ○ No, because (Open text)					
Q_{11}	Please rate the presented code regarding the following questions:					
$Q_{11\text{-}1}$	How easy was it to understand this code?					
$Q_{11\text{-}2}$	How easy would it be to maintain this code?					
$Q_{11\text{-}3}$	How easy would it be to extend this code?					
$Q_{11\text{-}4}$	How easy would it be to detect bugs in this code?					
	For each a Likert scale: ○ very hard ○ hard ○ easy ○ very easy					

After performing the tasks in Q_8 and Q_9, we asked our participants in Q_{10} and Q_{11} about their preferences regarding each code example. To this end, we first asked for a simple yes / no assessment of the appropriateness of the CPP directives, providing an option to explain why the participants did not consider the directives appropriate. In a second assessment, we asked our participants to refine that initial assessment on four-level Likert scales regarding four typical software development activities: program comprehension, maintenance, extension, and bug detection. This way, we aimed to identify whether specific styles of CPP use (e. g., disciplined versus undisciplined directives) are perceived positive regarding one activity, but negative regarding another.

E. Selection of Participants

Most empirical studies in software engineering have limited population sizes, resulting in limited external validity [36]. To tackle this issue, we contacted C developers from several open-source projects hosted on GitHub whose e-mail addresses were publicly available. We selected the projects based on previous works of Liebig et al. [20] and Medeiros et al. [25], and selected



(a) Task 1 (Q_8): Which of the following statements is true?

(b) Task 2 (Q_9): When would line <x> be executed?

Fig. 2: Correctness of our participants' responses for both program comprehension tasks.

additional ones that were trending on GitHub in October 2018 (e. g., redis,[2] FFmpeg[3]). Using this procedure, we aimed to mitigate sampling and coverage biases. We invited 7,791 developers of which 1,117 started and 521 finished our study (~7 %). As the minimum sample size (however large the overall population) to achieve a confidence level of 95 % is 385, we mitigated external threats and our results are of high confidence.

IV. Results

In this section, we report the results of our study with respect to the effect of the refactorings we applied. To this end, we report the details of our *participants' background,* the results for our *two research questions,* and *summarize* our observations. We provide an overview of all statistical tests we used to test our observations and the corresponding results in Tbl. III.

A. Participants' Background

Overall, 521 participants completed our study, with an almost even split between S1 and S2 (260 vs. 261 responses). Next, we analyze our participants' background based on the first seven questions in our study (cf. Tbl. II). Afterwards, we compare their responses for the baseline example (`vim18`). Based on these comparisons, we assess whether both study versions can be compared without introducing bias.

Considering their age (Q_1), most participants of S1 are 32 to 42 years old, and most of S2 are 27 to 42 years old. For both groups, median and mean are identical (37 and 36 years). The majority of our participants are male (Q_2), with S1 involving three females, one other, and 16 who preferred not to tell, while S2 involves eight females, five others, and nine who preferred not to tell. For their general programming experience (Q_3), most participants in S1 stated between 11 and 25 years; in S2, most stated between 12 and 25 years. The median and mean in both groups are identical (20 years). The C/C++ programming experience (Q_4) is almost equal between both groups, with most participants stating 8 to 20 years of experience (median and mean are 15 years). Concerning their roles (Q_5), most of our participants stated that they work as developers (S1: 250, S2: 249), while considerably fewer participants selected team manager (S1: 76, S2: 69), project manager (S1: 57, S2: 57), and quality assurance (S1: 40, S2: 40). As we allowed multiple answers, these numbers do not add up to 100 %. Our participants stated to have worked on a large variety of open-source projects (Q_6), including Linux (kernel and distributions), PostgreSQL, and OpenSSL. The average self-assessment of the participants' C/C++ programming skills (Q_7) is "advanced" (3.32 for S1 and 3.29 for S2).

Each group comprises participants with varying experience levels and, due to their roles, with different perspectives. Overall, however, the demographics of the two groups are highly similar. Consequently, we do not need to control for development experience and argue that our participants' demographics do not threaten the results of our study.

Our second control mechanism was our baseline example (`vim18`), which was identical in both studies. We show the results of the program comprehension tasks in Fig. 2, the appropriateness ratings in Fig. 3, and the general comprehensibility ratings in Fig. 4. Without going into detail, we can see that both groups performed similarly regarding the correctness of solving our two program comprehension tasks and had similar opinions of the baseline example. This indicates that there are no biases or imbalances between both groups, which, in turn, allows us to compare the results of both study versions.

B. *RQ$_1$*: Objective Correctness

We show the results for the first comprehension task (Q_8) in Fig. 2a and for the second comprehension task (Q_9) in Fig. 2b. For each example, we display how many participants solved each task correctly, incorrectly, or did not know the solution. Note that for `vim15` (S1) in Task 1, the 5 and the 1 are actually two numbers (incorrect and "don't know," respectively), even though they look like a 51 in the figure.

Observation$_1$: For Comprehension Task 1, we observe only marginal differences (Q_8). In Fig. 2a, we can see that our data indicates *almost no difference* between the original and refactored code regarding the correctness for Task 1. For all code examples, the amount of correct, incorrect, and "don't know" answers is almost identical between the code versions—with slightly fewer correct answers for the refactored examples.

[2]https://github.com/antirez/redis
[3]https://github.com/FFmpeg/FFmpeg

TABLE III: Statistical test results for our observations regarding the effect of refactoring CPP directives.

ID	Measure	Observation	Test	sig.	Effect	Size	Reason
Q_8	Comprehension Task 1	O_1	Fisher Exact Test	0.18	(negative tendency)		Not significant
Q_9	Comprehension Task 2	O_2	Fisher Exact Test	< 0.001	Negative	OR=0.74	61 % vs. 54 % correct
Q_{10}	Appropriateness	O_3	Fisher Exact Test	< 0.001	Positive	OR=1.60	52 % vs. 64 % positive ratings
Q_{11}	Code Quality	O_4	Wilcox & Cliff's Delta	< 0.05	(positive tendency)	0.05 – 0.07	Negligible

However, it seems that `vim13` is particularly difficult, as it received significantly fewer correct answers (for both versions of the code) than the other code examples.

Observation$_2$: For Comprehension Task 2, we see a slight tendency towards negative effects of refactoring (Q_9). In contrast to Task 1, our data reveals *slightly fewer* correct answers to Task 2 from participants working on the refactored code compared to participants working on the original code. The only exception is `vim13`, for which the number of correct answers was equally low for the original and the refactored code. Moreover, if we ignore correct and "don't know" answers, and only consider the incorrect answers, our data reveals that for *all* code examples, participants performed better on the original code compared to the refactored code. As we described in Sec. III, Task 2 was arguably more challenging to solve, as our participants had to "configure" the example (i. e., specify CPP macros) so that a particular line is executed. The increased difficulty is directly visible, as our participants performed considerably worse on all examples, except for `emacs12` in S1.

Hypothesis Testing. To test whether our observations are statistically significant, we used Fisher's exact test and the R statistics suite [12], [32]. With this test, we can determine how likely it is that our observations are merely the result of chance. Consequently, our *null hypotheses* are that ($H0_1$) participants perform equally well for Task 1, and that ($H0_2$) they perform equally well for Task 2.

As we show in Tbl. III, the p-value for Observation$_1$ is 0.18. Consequently, our observation may be purely accidental and we cannot reject $H0_1$. By contrast, the p-value for Observation$_2$ is < 0.001, which means that we can reject $H0_2$. To determine the effect size, we computed the average ratios of correct answers for the original and the refactored code examples, noting that the odds of correct answers decrease from 595:381 to 532:462, which amounts to an *odds ratio (OR)* of 0.74. In other words, the percentage of correct answers drops by 7 percent points, from 61 % to 54 %. The negative trend persists even if we calculate more pessimistically and consider "don't know" answers as incorrect (p=0.006, OR=0.78). While these effects may be too small to conclude that our refactorings actually hurt program comprehension, they at least suggest that the refactorings were *not beneficial.*

C. *RQ$_2$: Subjective Preference*

In Fig. 3, we show the subjective preference ratings of our participants concerning the appropriateness of the CPP directives in each example (Q_{10}). We can see that our participants considered `vim13` to comprise the least appropriate directives. This example was also the most challenging one during the program comprehension tasks.

In Fig. 4, we show the responses for question $Q_{11\text{-}1}$, in which our participants rated the ease to comprehend the examples on a Likert-scale. For simplicity, we omit the remaining plots for maintenance, extension, and bug detection ($Q_{11\text{-}2}$ through $Q_{11\text{-}4}$), because the responses were very similar. Interestingly, multiple examples are considered to be harder to comprehend, despite achieving a similar rating in terms of properly used CPP directives. For instance, `emacs11` and `emacs12` as well as their refactored counterparts are considered as similarly appropriate in terms of CPP use. However, `emacs11` and `emacs11_R` are both considered far more difficult to understand than `emacs12` and `emacs12_R`—which is in line with the results of comprehension Task 2, but not Task 1 (cf. Fig. 2).

Observation$_3$: Regarding the appropriateness of CPP directives, we observe a slight preference towards refactored code (Q_{10}). Overall, our participants generally consider the refactored CPP directives to be *more appropriate* than their original counterparts (cf. Fig. 3). The sole exception is `emacs11`, for which the refactored and original code performed almost identically. For `vim15`, we find the largest differences with ratings rising from 45 % to 70 %. Moreover, among the four examples in which less than 50 % of our participants considered CPP use appropriate, only `vim13_R` (most challenging during Q_8 and Q_9) is refactored.

Observation$_4$: For comprehension, maintenance, extending, and bug fixing, we observe a marginal difference (Q_{11}). Similar to Observation$_3$, we find (cf. Fig. 4 for comprehension) that our participants rate two refactored examples *marginally better*. We remark that this represents the averaged results, combining the ratings from $Q_{11\text{-}1}$ through $Q_{11\text{-}4}$. However, our participants also consider one original code example (`emacs11`) as marginally better compared to its refactored counterpart, and one as equal (`emacs12`).

Hypothesis Testing. To test whether our observation regarding

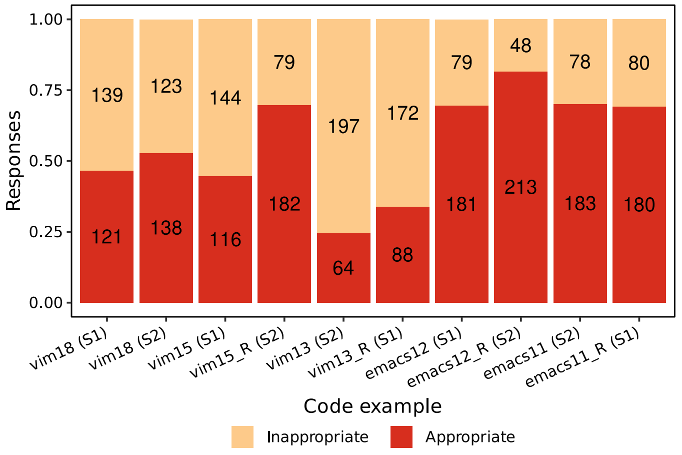


Fig. 3: Subjective rating of CPP directives use (Q_{10}).

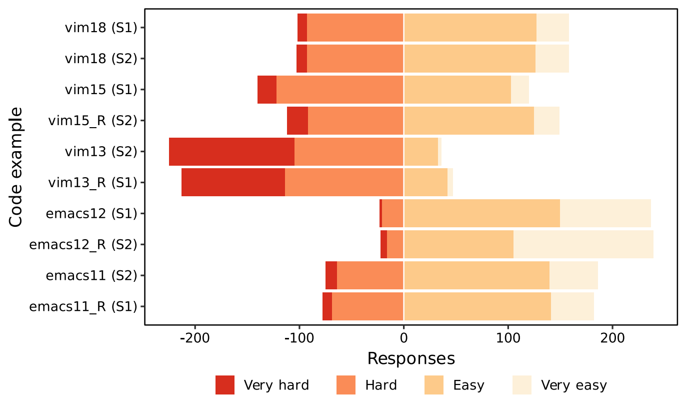


Fig. 4: Subjective rating of comprehensibility ($Q_{11\text{-}1}$).

appropriateness (Q_{10}) is significant, we applied the same procedure as for the comprehension tasks (cf. Sec. IV-B). The corresponding *null hypothesis* is that ($H0_3$) the proportions of participants rating a code example as appropriate are the same for the original and the refactored code.

For the subjective ratings of Q_{11}, we applied the Wilcoxon-Mann-Whitney U test as the significance test and Cliff's delta as the effect size measure. The *null hypothesis* is that ($H0_4$) the rating distributions regarding understandability, maintainability, extensibility, and ease of bug detection are the same for the original and the refactored code.

Considering Observation$_3$, Fisher's test indicates a significant p-value of < 0.001, and we calculated that the odds of favorable ratings rise from 544:489 (for the original code) to 663:379 (for the refactored code). This is an OR of 1.60 and is equivalent to a 12 percent points rise in favorable ratings. Therefore, we reject $H0_3$ and can assume that our refactorings improved the perceived appropriateness of CPP use. For Observation$_4$, in turn, the U test indicates significance with a p-value < 0.05 (cf. Tbl. III). However, the effect sizes for all four sub-questions are negligible (Cliff's delta ranges from 0.05 to 0.07). In other words, even though we can reject $H0_4$, our participants' preference for the refactored code is hardly noticeable when they judge CPP use *and* the underlying C code as a whole.

D. Summary

In summary, we observe that our participants preferred the refactored examples, considering them to be more appropriate and helpful concerning program comprehension. At the same time, we observe that our participants gave slightly more correct answers for the original code when answering questions regarding its configurability. The differences regarding both measures, subjective preference and objective correctness, are small, but they are statistically significant. Consequently, we find a discrepancy in the actual correctness versus preference regarding program comprehension in the context of CPP use. This result indicates a surprising dilemma that has not been identified in previous works, challenging established beliefs in academia and practice. We discuss our results in more detail within the next section.

V. Discussion

In this section, we first *answer our two research questions*. Based on these answers, we discuss the *acceptance of refactorings* for CPP directives and the dilemma between *correctness and preference* we identified for our participants.

A. Answering our Research Questions

In both comprehension tasks, we observed a tendency towards worse (or at least, not better) results on the refactored examples. For Task 1, the differences regarding correctness proved to be insignificant, but for Task 2 they were significant with an effect size of OR=0.74. We therefore answer $\mathbf{RQ_1}$ as follows:

> $\mathbf{RQ_1}$**:** Our refactorings of CPP directives *failed* to facilitate *objective* program comprehension correctness.

Most of our refactorings were *extract alternative function* and *discipline directives* refactorings, with which we targeted overly fine-grained and undisciplined directives. The literature has repeatedly documented that professional developers see both of these issues as problematic [24], [25], [27], which is why we expected our refactorings to be beneficial. To our surprise, we found the opposite, despite using particularly smelly code.

Regarding $\mathbf{RQ_2}$, our findings were more in line with the evidence and the prevailing opinion in the literature. In particular, our participants rated CPP use in the refactored examples as clearly preferable in Q_{10} and expressed a slight preference in Q_{11}. Thus, our answer to $\mathbf{RQ_2}$ is:

> $\mathbf{RQ_2}$**:** Our refactorings of CPP directives *slightly improved the subjective preference* of the source code, especially regarding the appropriateness of CPP use.

The rating differences were more pronounced for Q_{10} than for Q_{11}, and we are unsure why. A possible explanation is that we specifically asked about CPP use in Q_{10}, but asked about quality in general in Q_{11}. Therefore, our participants may have taken other factors, such as the complexity of the underlying C code, into account when answering Q_{11}. In fact, it appears that developers mostly ignore CPP directives when judging the quality of a piece of code.

Question Q_{11} had four sub-questions, in which our participants had to rate the ease of *comprehension, maintenance, extension,* and *bug detection.* We only showed the responses regarding comprehension in Fig. 4, because the responses regarding the other three aspects were distributed virtually identically. The data from the survey by Medeiros et al. [25] exhibits a similar trend: Irrespective of whether the participants were asked to rate understandability, maintainability, or fault-proneness, they gave similar ratings for each aspect. Their data and ours suggests that practitioners are either unable or unwilling to distinguish between different aspects of program comprehension and maintenance. Further research is needed to clarify whether awareness is the problem or whether the distinction is irrelevant in practice.

B. Acceptance of Refactorings

In preparing the examples for our study, we mainly performed two refactorings, *extract alternative function* and *discipline directives*. From the quantitative results we presented in Sec. IV-C, we can infer that our participants generally see these refactorings as beneficial. However, there were also some unexpected results, such as `emacs11_R`, which our participants considered to be just as good or bad as `emacs11`. To gain deeper insights into our participants' reasoning, we analyzed the qualitative comments given in Q_{10}.

We performed *extract alternative function* refactorings on three examples: `vim15_R`, `emacs11_R`, and `emacs12_R`. Given the existing evidence [25], we expected all examples to receive better ratings than their original counterparts, but surprisingly, this only happened for `vim15_R`. We believe the reason to be that the EMACS examples are shorter than the VIM examples. Moreover, their underlying C code is less complex. Due to these factors, they were already so easy to understand that *extract alternative function* brought no further improvement. Some participants even voiced critical remarks:

> **On `emacs11_R`**
> "[T]he function definitions are duplicated. This can confuse static analysis tools, but worse, it can confuse humans."

These insights make *extract alternative function* appear detrimental for short, easy functions. Whether this refactoring is advisable in other contexts, for example, for long, complex functions, should be investigated in future work.

In their free-text answers to Q_{10}, our participants clarified that they dislike great extents of fine-grained CPP directives inside function bodies:

> **On `vim18`**
> "Preprocessor macros should not be used like this, ever, because it makes the code hilariously and needlessly complicated and very hard to comprehend."

This dislike of certain usage patterns of CPP directives may also explain why the refactored `vim13_R` was criticized almost as much as its original counterpart, `vim13`:

> **On `vim13`**
> "I'm now considering giving up using VIM if that is how its code looks like."

> **On `vim13_R`**
> "When I considered use of preprocessor inappropriate in previous examples, now I think I was too harsh."

We created `vim13_R` from `vim13` by applying five *discipline directives* refactorings, but applied no other refactorings. Disciplining directives improved the ratings, which we expected given the evidence in the literature [24], [25], [27]. However, the improvement was small, causing `vim13_R` to still receive the second-worst ratings among all examples. This suggests that the main issue in `vim13` was not discipline, but the sheer number of CPP directives inside the function body. Since this issue was not addressed in the refactored version, the perceived code quality remained low.

We received 232 qualitative comments on `vim13` and 195 on `vim13_R`, which we analyzed to understand why these examples were criticized so heavily. Using open coding to group the comments, three main themes emerged, *understandability, complexity,* and *code quality*. Interestingly, the relative frequency with which these themes were mentioned differed between `vim13` and `vim13_R`: For `vim13`, understandability was the most frequent theme (49 % of the comments) and code quality the least frequent one (14 %). For `vim13_R`, however, code quality was the most frequent theme (53 %) and understandability was only the second-most frequent (34 %). In other words, disciplining directives led more participants to judge CPP use as appropriate (see Fig. 3), but it also shifted the perceived root of the problems from understandability to code quality. Thus, our answer to $\mathbf{RQ_2}$ (see Sec. V-A) must be refined:

> **Addition to $\mathbf{RQ_2}$:** Sometimes, refactoring improves the perceived quality of CPP directives, but at the expense of decreasing the perceived quality of the underlying C code.

C. Correctness versus Preferences

The most interesting and surprising result of our experiment are the contradicting answers to our research questions. As expected, our refactorings improved the subjective quality ratings, and thus our participants' preferences for the source code ($\mathbf{RQ_2}$). At the same time, our participants' objective program comprehension performance failed to improve ($\mathbf{RQ_1}$). In fact, it slightly worsened.

The objectively inferior correctness of solving comprehension tasks conflicts with the existing evidence, but only at first glance. Unlike us, no previous experiment has measured the comprehensibility of CPP directives both objectively *and* subjectively using the *same* code examples and the *same* developers. Moreover, the existing objective evidence against overly complex CPP directives (especially undisciplined directives) is largely based on experiments with novices. By contrast, most of our participants were highly experienced. It is therefore possible that our experimental setup and the reliance on professionals as subjects led us to obtain different results. We welcome future studies to replicate or refute our findings, and we argue that they are essential to understand and potentially resolve this dilemma.

Our results indicate that fine-grained or undisciplined CPP directives do not necessarily influence program comprehension of professional developers. In particular, our results show that developers' preferences of CPP use may contradict their ability to correctly solve program comprehension tasks. From this perspective, the importance of aligning CPP directives with syntactic code structures — something advised by professionals and academics alike — may have been overestimated. Similarly surprising results have been obtained in relation to code clones [13], code smells [37], and refactoring [40]. Our study is the first to show such a dilemma in the context of CPP use.

VI. Threats to Validity

Next, we discuss internal and external threats to validity [36].

A. Internal Validity

While we tried to not influence our subjects, we may have done so nevertheless. For instance, some questions may have implied critique towards the CPP. Similarly, some participants reported that they thought some questions were ambiguous. In particular, this affected Q_9 and Q_{10}, and may also be connected to the wording we used. Even though we formulated the questions with the goal of avoiding biases caused by strong wording, our wording may have caused ambiguity, which threatens the internal validity of our study.

Refactoring choices are always subjective. In preparing our examples, we followed advice from the literature and discussed all refactorings with multiple colleagues. Nevertheless, better refactoring choices may have been possible, which remains a threat to the internal validity of our study.

We aimed to minimize the time to complete our study. Still, the average completion time was half an hour, which is quite long and may have discouraged developers who were unwilling to spend that much time. A shorter study may have been an alternative, but it may have prevented us from observing, for example, the dilemma between correctness and preference. Thus, this issue remains a threat to the validity of our study.

B. External Validity

We controlled the external validity by using real-world examples and inviting a large number of participants (cf. Sec. III-E). In the end, we obtained enough participants to ensure that our results and statistical tests provide reasonable insights. We further controlled for our participants' experiences to ensure that we could compare between both studies. Nonetheless, some participants may have used some kind of external help, as we conducted an unsupervised study. Consequently, this issue may threaten the generalizability of our results.

Another threat are the examples that we used: They are from the same domain (open-source text editors), we refactored them ourselves, and we only used a small number of examples. However, research shows that industrial and open-source software is comparable considering the use of the CPP [11], and the refactorings we employed are based on established research (cf. Sec. II). We took further design decisions to improve the response rate (e. g., duration of the study, online study) and ensure that we can compare both groups (e. g., distribution of the examples). These decisions are potential threats to the external validity of our study.

Additionally, our participants were generally not familiar with the code examples they worked on. They may have performed better on known code and may have judged code quality differently [17]. While this threatens the external validity of our study, it improves the internal validity since all participants had the same degree of knowledge about the code.

Finally, we are aware that several human factors can threaten the external validity of our study. We aimed to mitigate some factors with our study design, for example, controlling our participants' experience and avoiding learning biases by using different examples. However, other factors are much harder to control. For instance, some participants may have been less motivated. Moreover, the cognitive processes of program comprehension are highly dependent on the individual developer. We partly mitigated these threats by involving experienced developers, reducing the workload, and removing unfinished responses. While especially the detailed responses to our free-text questions indicate that our participants were motivated, these threats remain a potential bias.

VII. CONCLUSIONS

In this paper, we reported the results of a large-scale empirical study in which we analyzed how refactoring CPP directives impacts program comprehension. For this purpose, we created an online study consisting of five real-world code examples with particularly smelly CPP directives. Four of those examples were refactored to reduce the complexity of the directives, and we deployed our study such that each participant worked on two original examples and on two refactored examples. The first example was the same for all participants, allowing us to establish that the groups of participants were comparable in terms of programming experience, age, and other individual factors. For every example, our participants solved two program comprehension tasks and provided their opinions on the quality of the code, combining objective measurements with subjective preferences. We distributed our study among open-source developers and received 521 responses. To the best of our knowledge, this is the first large-scale study to combine objective and subjective perspectives, thus considerably extending previous research. We derived the following insights:

- Regarding correctness, our participants failed to perform better on the refactored code, despite previous evidence suggesting that this code should be considerably less smelly. Indeed, our participants performed slightly worse.
- Our participants preferred the way that CPP directives were used in the refactored examples over the original ones.
- Our results contradict each other and previous findings regarding the usefulness of refactoring CPP directives.
- Refactoring CPP directives is a trade-off with the quality of the underlying code, meaning that refactoring smelly directives may result in a net decrease in overall quality.

Our results indicate a surprising dilemma that has not been reported in previous work. This dilemma challenges established beliefs on refactoring certain CPP usage patterns, such as undisciplined directives, to improve program comprehension.

In future work, we plan to analyze the dilemma we identified to shed light on its causes. This may involve other forms of program comprehension, such as dataflow analysis, which the present study did not cover. Also, it seems promising to empirically investigate the interplay of CPP directives and code quality, as our results suggest that one may be traded against the other. Finally, we argue that we need to better understand what CPP-related refactorings are actually helpful in what situations.

Acknowledgments. This research has been supported by the DFG project EXPLANT (SA 465/49-3). We thank our anonymous reviewers and the participants of the FOSD meeting 2019 for their feedback on early stages of this work, and we thank the participants of our study for supporting our research.

References

[1] I. Abal, C. Brabrand, and A. Wąsowski, "42 variability bugs in the Linux kernel: A qualitative analysis," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 421–432.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[3] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 12, pp. 1146–1170, 2002.

[4] J.-M. Favre, "Preprocessors from an abstract point of view," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 1996, pp. 329–338.

[5] ——, "Understanding-in-the-large," in *Proceedings of the International Workshop on Program Comprehension (IWPC)*. IEEE, 1997, pp. 29–38.

[6] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the #ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[7] W. Fenske, S. Schulze, D. Meyer, and G. Saake, "When code smells twice as much: Metric-based detection of variability-aware code smells," in *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2015, pp. 171–180.

[8] W. Fenske, S. Schulze, and G. Saake, "How preprocessor annotations (do not) affect maintainability: A case study on change-proneness," in *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2017, pp. 77–90.

[9] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel, "Do #ifdefs influence the occurrence of vulnerabilities? An empirical study of the Linux kernel," in *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2016, pp. 65–73.

[10] P. Gazzillo and R. Grimm, "SuperC: Parsing all of C by taming the preprocessor," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 323–334.

[11] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel, "Preprocessor-based variability in open-source and industrial software systems: An empirical study," *Empirical Software Engineering*, vol. 21, no. 2, pp. 449–482, 2016.

[12] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *Journal of Computational and Graphical Statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[13] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[14] C. Kästner and S. Apel, "Virtual separation of concerns – A second chance for preprocessors," *Journal of Object Technology*, vol. 8, no. 6, pp. 59–78, 2009.

[15] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2011, pp. 805–824.

[16] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a better understanding of software features and their characteristics: A case study of Marlin," in *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, 2018, pp. 105–112.

[17] J. Krüger and R. Hebig, "What developers (care to) recall: An interview survey on smaller systems," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020.

[18] J. Krüger, K. Ludwig, B. Zimmermann, and T. Leich, "Physical separation of features: A survey with CPP developers," in *Proceedings of the ACM Symposium on Applied Computing (SAC)*. ACM, 2018, pp. 2042–2049.

[19] D. Le, E. Walkingshaw, and M. Erwig, "#ifdef confirmed harmful: Promoting understandable software variation," in *Proceedings of the Symposium on Visual Languages and Human Centric Computing (VL/HCC)*. IEEE, 2011, pp. 143–150.

[20] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.

[21] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 191–202.

[22] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, "A quantitative analysis of aspects in the eCos kernel," in *Proceedings of the European Conference on Computer Systems (EuroSys)*. ACM, 2006, pp. 191–204.

[23] K. Ludwig, J. Krüger, and T. Leich, "Covert and phantom features in annotations: Do they impact variability analysis?" in *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 2019, pp. 218–230.

[24] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi, "The discipline of preprocessor-based annotations – Does #ifdef TAG n't #endif matter," in *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 297–307.

[25] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 495–518.

[26] F. Medeiros, M. Ribeiro, and R. Gheyi, "Investigating preprocessor-based syntax errors," in *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013, pp. 75–84.

[27] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca, "Discipline matters: Refactoring of preprocessor directives in the #ifdef hell," *IEEE Transactions on Software Engineering (TSE)*, vol. 44, no. 5, pp. 453–469, 2018.

[28] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi, "An empirical study on configuration-related issues: Investigating undeclared and unused identifiers," in *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2015, pp. 35–44.

[29] J. Melo, C. Brabrand, and A. Wąsowski, "How does the degree of variability affect bug finding?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 679–690.

[30] R. Muniz, L. Braz, R. Gheyi, W. Andrade, B. Fonseca, and M. Ribeiro, "A qualitative analysis of variability weaknesses in configurable systems with #ifdefs," in *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 2018, pp. 51–58.

[31] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki, "The shape of feature code: An analysis of twenty C-preprocessor-based systems," *Software & Systems Modeling (SOSYM)*, vol. 16, no. 1, pp. 77–96, 2017.

[32] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, 2018. [Online]. Available: https://www.R-project.org

[33] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending studies on program comprehension," in *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 308–311.

[34] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter? A controlled experiment," in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2013, pp. 65–74.

[35] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1299–1334, 2014.

[36] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 9–19.

[37] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 8, pp. 1144–1156, 2013.

[38] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with C News," in *Proceedings of the USENIX Technical Conference*. USENIX Association, 1992, pp. 185–197.

[39] P. Toft, D. Coleman, and J. Ohta, "A cooperative model for cross-divisional product development for a software product line," in *Proceedings of the International Software Product Line Conference (SPLC)*. Springer, 2000, pp. 111–132.

[40] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad,” in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 403–414.

[41] A. M. Vans, A. von Mayrhauser, and G. Somlo, “Program understanding behavior during corrective maintenance of large-scale software,” *International Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 31–70, 1999.