Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment

Sandro Schulze
Technische Universität Braunschweig
Braunschweig, Germany
s.schulze@tu-braunschweig.de

Jörg Liebig, Janet Siegmund,* Sven Apel
Universität Passau
Passau, Germany
{joliebig,siegmunj,apel}@fim.uni-passau.de

Abstract

The C preprocessor (CPP) is a simple and language-independent tool, widely used to implement variable software systems using conditional compilation (i.e., by including or excluding annotated code). Although CPP provides powerful means to express variability, it has been criticized for allowing arbitrary annotations that break the underlying structure of the source code. We distinguish between *disciplined* annotations, which align with the structure of the source code, and *undisciplined* annotations, which do not. Several studies suggest that especially the latter type of annotations makes it hard to (automatically) analyze the code. However, little is known about whether the type of annotations has an effect on program comprehension. We address this issue by means of a controlled experiment with human subjects. We designed similar tasks for both, disciplined and undisciplined annotations, to measure program comprehension. Then, we measured the performance of the subjects regarding correctness and response time for solving the tasks. Our results suggest that there are no differences between disciplined and undisciplined annotations from a program-comprehension perspective. Nevertheless, we observed that finding and correcting errors is a time-consuming and tedious task in the presence of preprocessor annotations.

Categories and Subject Descriptors [*Software and its Engineering*]: Preprocessors; [*Software and its Engineering*]: Software product lines; [*General and reference*]: Experimentation

Keywords variability; C preprocessor; controlled experiment; program comprehension; disciplined annotations

1. Introduction

The *preprocessor* CPP, developed over 40 years ago, is widely adopted in the practice of software development to introduce variability in software systems [6, 26]. Being a simple and language-independent text-processing tool, CPP provides powerful and expressive means to implement variable source code [26]. Programmers use preprocessor annotations (e.g., #ifdef directives) to implement optional and alternative code fragments. Since CPP is language-independent, programmers can use annotations at a fine grain, for instance, by annotating single tokens, such as an opening bracket. CPP is often criticized for this capability, as fine-grained annotations are a major source of errors. For instance, practitioners report from maintainability and understandability problems with arbitrary preprocessor usage [3, 34]. Furthermore, preprocessor usage, especially at a fine grain, hinders tool support for code analysis or restructuring tasks [7, 8, 14–16]. Hence, the source of all problems is the lack of discipline of annotations (i.e., their usage at a fine grain), and how programmers understand code in their presence.

In earlier work, we analyzed the discipline of annotations and distinguished *disciplined* and *undisciplined* annotations [27]. Disciplined annotations align with the underlying structure of the source code by targeting only code fragments that belong to entire subtrees in the corresponding abstract syntax tree. For example, we consider an annotation enclosing a whole function definition to be disciplined. In contrast, undisciplined annotations include arbitrary annotations of code fragments, for instance, an annotation of a single function parameter. One reason for why we consider the latter annotation undisciplined is that such fine-grained annotations are difficult to refactor using tool support [8, 15].

Another reason is experience from practice: Some software developers are aware of problems related to CPP and introduced coding guidelines for preprocessor usage. For instance, one guideline for developers of the Linux kernel for using the CPP states:[1]

> *Code cluttered with ifdefs is difficult to read and maintain. Don't do it. Instead, put your ifdefs in a header, and conditionally define static inline functions, or macros, which are used in the code.*

In fact, this guideline advises developers to use disciplined instead of undisciplined annotations, such as annotating only entire functions instead of annotating parts of the function definition. This and similar guidelines express long-term experiences and opinions of developers. However, while these guidelines rely on experience, they are not the result of a sound empirical investigation. Hence, several questions regarding the interaction of programmers with annotated code are still open:

1. Do programmers understand code with disciplined annotations better than code with undisciplined annotations?
2. Are maintenance tasks, such as adding or modifying code, more difficult or even more error-prone in the presence of undisciplined annotations than with disciplined annotations?

* Janet Siegmund has published previous work as Janet Feigenspan.

[1] see /Documentation/SubmittingPatches in the Linux source

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2373-4/13/10. . . $15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2517208.2517215

3. How does the discipline of annotations influence the detection and correction of errors?
4. Generally speaking, are there differences in the programmers' performance or correctness (regarding the tasks) with respect to disciplined and undisciplined annotations in source code?

To answer these questions, we conducted a controlled experiment with 19 undergraduate students from the University of Magdeburg. We designed seven different tasks that aim at understanding and maintaining annotated source code, including the detection and correction of errors.

In a nutshell, the results of our experiment do *not* support the assumption that the discipline of annotations has an observable effect on comprehension of annotated code. For both types of annotations, the respective groups performed similarly regarding correctness and response time. However, we made the general observation that detecting and fixing errors in the presence of annotations is a tedious and time-consuming task with only minor success.

The remainder of the paper is organized as follows: In Section 2, we introduce preprocessor annotations and how they are used for expressing variability. In Section 3, we describe our experimental setting. We present the results of our experiment and interpret them in Section 4. In Section 5, we discuss threats to validity, followed by a discussion of related work (Section 6). Finally, we present our conclusions and suggestions for future work in Section 7.

2. Preprocessor Annotations in Action

The C preprocessor CPP is a simple text-processing tool that provides metaprogramming facilities, and that is used by programmers to implement variable source code. Programmers use it to mark optional or alternative code fragments in the programming language of their choice (host language), and to include and exclude these code fragments on demand (controlled by macro constants).[2] Since the preprocessor is a language-independent tool and works on the basis of tokens of the target language, it allows programmers to annotate variable source code at any level of granularity. For example, in the programming language C, annotations may be on single tokens (e.g., an opening or closing bracket), expressions, statements, or type definitions. In earlier work, we analyzed CPP's annotation capabilities and distinguished between disciplined and undisciplined annotations [27]:

> In C, annotations on one or a sequence of *entire functions* and *type definitions* (e.g., struct) are disciplined. Furthermore, annotations on one or a sequence of *entire statements* and annotations on *elements inside type definitions* are disciplined. *All other annotations are undisciplined.*

Disciplined and undisciplined annotations are in the center of a larger discussion about expressiveness, replication, and comprehension of source code (Figure 1). While the first two aspects have been discussed already elsewhere [27, 31], we focus on program comprehension, which has not been investigated yet systematically (to the best of our knowledge).

We explain the trade-off between the aforementioned properties by means of a variable stack implementation (with or without synchronization). To this end, we show the respective source code of a disciplined and an undisciplined version in Figure 2. The implementation of synchronization using undisciplined annotations requires to add a function parameter (Figure 2a; Line 4), to extend an existing expression (Figure 2a; Line 9), and to add statements in the middle of a function body (Figure 2a; Lines 14 and 18). By using undisciplined annotations, we can omit code replication, but have

[2] Although the preprocessor CPP is typically part of C/C++ compiler infrastructures, it can be used with any programming language.

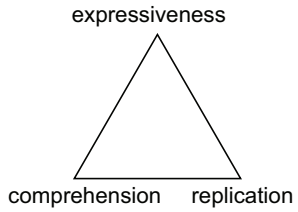


Figure 1. Trade-off between expressiveness, comprehension, and replication

to use four different annotations. In contrast, the disciplined stack implementation (Figure 2b) has only two annotations (#ifdef and #else branches), but contains several replicated code fragments: the declaration of function push with the object parameter o (Figure 2b; Lines 3 and 14), the null pointer check and the return statement (Figure 2b; Lines 5 and 14), the assignment to the array (Figure 2b; Lines 8 and 16), and the function call fireStackChanged (Figure 2b; Lines 10 and 17).

```
1  class Stack {
2    void push(Object o
3  #ifdef SYNC
4    , Transaction txn
5  #endif
6    ){
7      if (o==null
8  #ifdef SYNC
9      || txn==null
10 #endif
11     )
12     return;
13 #ifdef SYNC
14     Lock l=txn.lock(o);
15 #endif
16     elementData[size++] = o;
17 #ifdef SYNC
18     l.unlock();
19 #endif
20     fireStackChanged();
21   }
22 }
```

(a)

```
1  class Stack {
2  #ifdef SYNC
3    void push(Object o,
4        Transaction txn)
5    {
6      if (o==null || txn==null)
7        return;
8      Lock l = txn.lock(o);
9      elementData[size++] = o;
10     l.unlock();
11     fireStackChanged();
12   }
13 #else
14   void push(Object o) {
15     if (o==null)
16       return;
17     elementData[size++] = o;
18     fireStackChanged();
19   }
20 #endif
21 }
```

(b)

Figure 2. Undisciplined (a) and disciplined (b) stack implementation

Generally, undisciplined annotations can *always* be transformed into disciplined ones (at the extreme end by replicating a whole source file), and vice versa. Consider the undisciplined annotation of the subexpression '|| txn==null' in our stack implementation (Figure 2; Line 9).[3] To transform this undisciplined annotation, we lift the annotation to the upper if-statement. The result are two differently annotated if-statements, one with the subexpression '|| txn==null' and one without (cf. Figure 2). Both annotations are disciplined according to our definition. The result are two disciplined annotations that form an alternative [27].

In general, by using undisciplined annotations, programmers are able to reduce the amount of code replication. Although recent studies show that code replication is not intrinsically harmful [17, 21], it can be a source of errors and influence the underlying software systems and its development in a negative way, such

[3] The different annotations of the undisciplined stack implementation overlap when we transform them into disciplined annotations. Therefore, in our case, we only have two large annotations (#ifdef – #else – #endif; Lines 7, 10, and 13) instead of multiple, more fine-grained annotations.

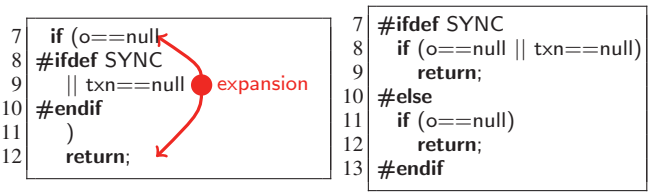


Figure 3. Transformation of an undisciplined annotation into a disciplined one

as increased maintenance effort or error propagation [2, 20]. By using undisciplined annotations, differences in code fragments can be factored out at a fine grain. However, reducing the amount of code replication comes at the price of introducing undisciplined annotations, which are considered to be more difficult to understand. For instance, preprocessor annotations obfuscate the source code and make it difficult to differentiate and locate source code [11, 22].

Beside the influence on program comprehension, an undisciplined use of the preprocessor has further implications. For instance, analysis tools (e.g., data-flow analysis) or transformation tools (e.g., source-code refactoring) require a structural representation of the code in form of an abstract syntax tree. With undisciplined annotations, such a representation is difficult to create, because fine-grained annotations, such as an annotated opening bracket, cannot be represented in terms of the abstract syntax of the host language. The reason is that single tokens, such as an opening bracket, may not have counterparts in the abstract syntax tree. As a consequence, many current IDEs struggle with software projects that make use of preprocessors. Typically, in such IDEs, the preprocessor annotations are removed, which causes a loss of variability information. To shed light on the preprocessor discipline, in previous work we analyzed the usage of disciplined and undisciplined preprocessor annotations in 40 software projects from different domains and sizes [27]. Except for one small project, we found undisciplined annotations in all projects. In summary, undisciplined annotations sum up to 16 % of all annotations.

Although the major part of preprocessor usage is disciplined, undisciplined annotations are still frequently used, despite their disadvantages. This raises the question of whether programmers perform differently using undisciplined or disciplined annotations. More precisely, are there differences in program comprehension for common programming tasks, such as maintenance, with respect to the discipline of annotations? To answer this question, we designed a controlled experiment. Next, we give a detailed description of the experiment and the material we used.

3. Experiment

By means of an experiment, we evaluate whether the kind of annotation (disciplined vs. undisciplined) has an influence on program comprehension. To this end, we let subjects solve programming tasks on several open-source systems and analyzed the correctness and response time. According to the work of Dunsmore et al., these tasks can be categorized as *maintenance* and *mental-simulation* tasks, both requiring program comprehension [5]. Next, we give a detailed description of our experimental setting, whereas we present and discuss the result in Section 4. Both sections are structured according to the guidelines of Jedlitschka et al. [19].

3.1 Objectives

The main objective of our experiment is to evaluate whether the discipline of preprocessor annotations has an influence on program comprehension. There is an ongoing debate about the discipline of preprocessor annotation, and the result is yet open. Some people argue in favor of disciplined annotations, because they ease automated analysis and thus tool support for the respective programs [6, 15]. Other people, in turn, prefer undisciplined annotations, because they provide flexibility, expressiveness, and avoid bloated code. As a matter of fact, undisciplined annotations are commonly used by professional developers [27]. Some developers argue that they have no problem with understanding their own code that contains undisciplined annotations.[4] While this may be reasonable for small, one-man software systems, it may become a problem in large systems, in which several developers are involved.

Due to these opposing positions regarding the discipline of annotations, we do not state a hypothesis in favor for a particular kind of discipline, but rather we formulate two research questions, which reflect the essence of the four open questions, which we posed in Section 1:

RQ.1 Does the discipline of annotations influence the correctness of program-comprehension and maintenance tasks?

RQ.2 Does the discipline of annotations influence the time needed to solve program-comprehension tasks?

Based on these research questions, we define two dependent variables: response time and correctness. To ensure, that these variables are not influenced or biased by other factors, we also have to control potential confounding parameters [33]. In Table 3.1 we show the five parameters we found to be most important to control for our experiment, together with the control technique we used to control them and the corresponding measurement.

Table 1. Confounding parameters, how we controlled them and which measurement we used

Conf. parameter	Control technique	Measurement
Motivation	Analyzed afterwards	Questionnaire
Difficulty	Analyzed afterwards	Questionnaire
Programming experience	Balancing	Questionnaire
Domain knowledge	Kept constant	—
Tool familiarity	Kept constant	Proprietary editor w. typical functions

We controlled the first two parameters, because different levels of motivation or an unbalanced difficulty between tasks may bias the results. We analyzed both parameters after completion of the tasks (participants had to complete a questionnaire), because it is not possible to evaluate these parameters in advance. Next, we controlled programming experience by forming two balanced groups based on a questionnaire (cf. Section 3.2). Finally, we had to control the level of domain knowledge (with respect to subject systems and participants), and how familiar the participants are with the tool that they used during the experiment.

3.2 Subjects

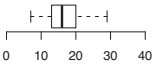
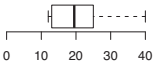
We recruited 19 undergraduate students from an operating-system course of the University of Magdeburg, Germany. As part of the lecture, the students had to implement a basic operating system, and thus were familiar with C/C++ and the CPP. However, the participants were neither aware of the different types (disciplined/undisciplined) of annotations nor the discussions about them. As a motivation, the participants had the chance to win one of several Amazon gift cards and could omit a mandatory exercise for participating in the experiment.

[4] This statement is an outcome of several discussions with professional C developers, e.g., Daniel M. German at PASED Summer School 2011.

Prior to the experiment, participants completed a questionnaire to measure their programming experience, which has been designed and evaluated carefully [10], based on state-of-the-art guidelines for empirical research. Within this Web-based questionnaire we asked the participants to estimate their programming experience regarding different languages, paradigms, and their participation in software projects. The participants had to assess their skills on a five-point Likert scale for each question [28]. In particular, participants had to respond on different questions by assessing their skills from 1 (low) to 5 (high). Finally, we computed the experience rank for each participant as a weighted sum based on the answers of the questionnaire. For this computation, we decided to give the experience with the C programming language a higher priority than for other languages, such as Java or Haskell, which is reflected by a higher weighting during computation. The reason is that, because the subject systems are written in C, knowledge of C is essential to obtain meaningful results.

Based on the results, we formed two homogeneous groups for our experiment, by applying a matching on the computed experience rank of the participants [18]. The goal of forming homogeneous groups was to have two comparable groups with a similar experience rank. As a result, we obtained groups with 8 and 11 participants, respectively (see Section 3.7 for explanation of the different group size). The smaller group had an experience rank of 20.8, on average (standard deviation: 6.89), and is referred to as *undisciplined group* in the remainder, because this group worked on the source code with the undisciplined annotations. Likewise, the other group had an experience rank of 16.8, on average (standard deviation: 5.26), and is referred to as *disciplined group*. Note that due to the different group size, the experience between both group differs. Nevertheless, it is still similar enough so that we can consider both groups to be comparable regarding their programming experience. We show the detailed experience rank for each participant of both groups in Table 2.

Table 2. Experience rank for disciplined and undisciplined group (descending order)

Participant	Rank discip. group	Rank undiscip group
#1	29	40
#2	27	28
#3	20	22
#4	20	19
#5	17	19
#6	16	14
#7	13	12
#8	13	12
#9	13	–
#10	10	–
#11	7	–
Distribution		

3.3 Material

To be as close as possible to real applications, we used real code from four open-source systems from different domains. Particularly, we used parts of *boa* (a web server), *dia* (a chart/diagram application), *irssi* (an IRC client), and *xterm* (a terminal emulator) as material. Due to their diversity (e.g., different domains, different programmers), we argue that these system are sufficiently representative for real-world software systems. However, since we aimed at analyzing program comprehension for different tasks and for different types of annotations, we had to prepare the code manually (e.g., removing files/lines of code). Otherwise, it would be infeasible for the participants to understand all the systems in detail, and to solve the respective tasks. Furthermore, by narrowing down the sample systems, we were able to emphasize the code that is of interest for our experiment and to mitigate side effects that may influence the result of our empirical study such as that participants sift through irrelevant code.

For selecting appropriate files for each task, we relied on our experience and former studies on preprocessor annotations [11, 27]. Consequently, we selected files that contain not only an average amount of annotated code (according to [26, 27]), but especially *undisciplined* annotations. Subsequently, we created a second version of each file by transforming undisciplined annotations to disciplined annotations using the expansion technique by Liebig et al. [27]. For instance, in our stack example in Figure 2b, we have disciplined annotations at the function level, but both annotated code fragments are replicated. Note that we explicitly avoided source code with mixed annotations, that is, disciplined and undisciplined annotations in the same place. Although such tangling may occur in real-world systems, it would render our results meaningless to some extent, because we could not measure *which* kind of annotation has an influence on our measurements (i.e., response time and correctness). Finally, we shortened some files, such that they fit to the time constraints of our experiment. This step of preparation was mainly initiated by feedback from our pre-tests with masters and PhD students, which we present in Section 3.4.

Next, we computed two code metrics: discipline of annotation and code clones (i.e., replicated code) [27, 30, 31], which we summarize in Table 3. We specifically computed the amount of code clones, because in recent studies we have shown that the discipline of annotations comes at the cost of replicated code (cf. Figure 1), which may influence the programmer's performance in maintenance tasks [31]. Additionally, code clones may affect program comprehension, because they may hinder the developer in building a mental model of the underlying code [24]. Regarding the discipline metric, we computed the number of disciplined annotations compared to all annotations in the system under study. As a result, we observed for the undisciplined version of the material, that the number of disciplined annotations differs from 33% to 91% with respect to all preprocessor annotations in the code (cf. Table 3).

3.4 Pilot Study

To assess whether our material is appropriate for our experiment (e.g., regarding time or difficulty), we conducted a pilot study, involving undergraduate and graduate students. Three students (1 master, 2 PhD) from the University of Passau and six students (4 master, 2 PhD) from the University of Magdeburg confirmed their participation. All of them worked with a similar setting as the participants in the experiment: They had to complete seven tasks within 90 minutes (to avoid fatigue effects). In contrast to the experiment, the pilot study took place at different times and different places (e.g., in office or at home). In addition to solving the tasks, we asked the subjects to make immediate comments on each task with pen & paper or within the form that is used to type in the solution for the respective task. Finally, we conducted interviews with each pilot tester to ask for her opinion regarding the material as well as the tasks. Based on the pilot study, we revised our material according to the comments of the pilot tester. In particular, we rephrased task descriptions that have been ambiguous to make them more understandable. Furthermore, for certain tasks, the pilot testers complained that the source code was too long for the given time. Hence, we shortened the respective files by removing parts that contain no preprocessor annotations.

Table 3. Summary of the material used for the tasks of the experiment, including annotation discipline and code clone metrics

Task	System	Version	SLOC D	SLOC UD	Discipline in % D	Discipline in % UD	Clones in % D	Clones in % UD	Task
T1	*boa*	0.94.13	1 405	1 404	100	90	5.9	5.9	Identifying all (different) preprocessor variables
T2	*xterm*	2.4.3	1 047	961	100	91	11.7	0.0	Determining the max. depth of `#ifdef` nesting
T3	*vim*	7.2	233	135	100	33	77.8	0.0	Determining the number of possible variants
T4	*vim*	7.2	606	475	100	41	0.0	0.0	Identifying code fragments that belong to a variant
T5	*irssi*	0.8.13	287	282	100	69	0.0	0.0	Add a new variant by modifying existing code
T6	*irssi*	0.8.13	457	447	100	86	0.0	0.0	Delete code that belongs to a given variant
T7-d	*xterm*	2.4.3	100	100	79	79	0.0	0.0	Correct an error
T7-u	*xterm*	2.4.3	100	100	79	79	0.0	0.0	Identify, whether an error occurs

D – disciplined group, UD – undisciplined group; All software systems are available on the Web: `http://freecode.com`.

3.5 Tasks

For our experiment, we created seven tasks, each of them related to one of two categories [5]: mental simulation or maintenance. Additionally, for each task, we provided the relevant material (source code), as explained in the previous subsection. Moreover, the tasks had a fixed order.[5]

The first four tasks T1–T4 correspond to mental simulation, each of them requires a grasp of how variability is expressed with preprocessor annotations. With these tasks, we measure how participants understand variability introduced by annotations. For instance, we asked the participants in task T1 to identify all the different preprocessor variables in the given piece of code. Another example is task T3, in which we asked participants for the number of possible variants of a certain function. Although such tasks may rarely occur explicitly in practice, programmers often face such tasks implicitly, for example, when trying to reproduce internals of source code (in our case, variability).

Tasks T5 and T6 (cf. Table 3) correspond to maintenance: We let participants modify and delete annotated code, which also requires understanding the respective source code. This may be necessary due to changed user requirements, which is quite common in software maintenance. For example, in task T6, participants had to remove all code that is related to a certain preprocessor variable.

Finally, with task T7 (related to mental simulation), we aim at discovering how developers detect and correct (syntax) errors in the presence of preprocessor annotations. This task is different, compared to the previous tasks, in two ways. First, the corresponding source code is identical for both the disciplined and undisciplined group. Second, the task itself is different for both groups. The reason is that, for disciplined annotations, syntax errors can be detected (by definition) *before* the preprocessing step of the CPP (e.g., by the parser) [23, 27]. Hence, with disciplined annotations, programmers do not need to detect syntax errors manually. In contrast, for undisciplined annotations, one can detect syntax errors automatically only after this preprocessing step. Hence, we focused solely on undisciplined annotations within task T7.

For the disciplined group, we provided the information (within the task description) that the code contains a syntax error and in which situation this error occurs (cf. T7-d in Table 3). Based on this information, the task was to rewrite the code to *fix the error*. In contrast, the task for the undisciplined group was to check whether the source code is syntactically correct for all configurations of preprocessor variables (i.e., to *detect the error*). We discuss the implication of this task design in Section 5. In Figure 4, we show the respective code snippet that contains the error.[6]

For all tasks, we created sample solutions in advance to compare the subjects' answers to it. This way, we aim at eliminating the possibility that the assessment of the solutions of the experiment is biased.

```
1  #if defined(__GLIBC__)
2  // additional lines of code
3  #elif defined(__MVS__)
4  result = pty_search(pty);
5  #else
6  #ifdef USE_ISPTS_FLAG
7    if (result) {
8  #endif
9    result = ((*pty = open("/dev/ptmx", O_RDWR)) < 0);
10 #endif
11 #if defined(SVR4) || defined(__SCO__) || \
12       defined(USE_ISPTS_FLAG)
13   if (!result)
14     strcpy(ttydev, ptsname(*pty));
15 #ifdef USE_ISPTS_FLAG
16   IsPts = !result;
17 }
18 #endif
19 #endif
```

Figure 4. Example of undisciplined annotation in *xterm* (task T7)

3.6 Execution

We conducted the experiment in November 2011 in a computer lab at the University of Magdeburg, with standard desktop machines and 19 inch displays. At the beginning of the experiment, we gave a short introduction to the experiment in the form of a presentation. Furthermore, we used PROPHET as tool infrastructure, which has been specifically developed for supporting program-comprehension experiments [10, 13]. PROPHET provides the basic functionalities of an Eclipse-like IDE, including as a file explorer, an editor with syntax highlighting, as well as a project and file search. Additionally, PROPHET has a dedicated dialog to show the task description to participants, and it provides a form for typing the respective answers. Beside this, we enabled source-code editing of the text editor for tasks T5 to T7.

With the help of PROPHET, we were able to track the activities of the participants and thus to use the resulting data for the analysis of the experiment. Basically, PROPHET records the answers and

[5] The concrete tasks, together with the source code, for both control groups are available on the Web: `http://www.fosd.net/experimentIfdef`.

[6] The error occurs in the case that the preprocessor variables `__GLIBC__` and `USE_ISPTS_FLAG` are defined.

time needed for each task. Furthermore, it logs the participants' activities while solving tasks. In particular, it logs different activities within the text editor, such as searching, scrolling, or editing. This, in turn, allows us to reason about peculiarities that we observe during the experiment and its analysis.

We conducted the experiment with a time limit of 90 minutes. If this limit was reached, we asked the participants to quit, but they were allowed to finish the task they were currently working on. We presented the tasks to participants in sequential order, one at a time. For each task, we recommended a time as a guideline for participants. However, participants were free to spent as much time as they wanted for an individual task. Finally, we asked participants about the difficulty and their motivation for each task using a questionnaire.

3.7 Deviations

During the execution of our experiment, two deviations occurred. First, two students took the experiment before all other participants (same day, but different time). Hence, they could have talked to other participants about the experiment, which may bias our results. However, they credibly assured that they did not disseminate any information about the experiment to the other participants. Second, three students did not complete the questionnaire prior to the experiment. Consequently, we could not assign them to any group in advance. Hence, these participants completed the questionnaire directly before the experiment. Subsequently, we randomly assigned them to a group by a coin toss. Nevertheless, we argue that this deviation does not influence our results for two reasons. First, the experience rank is similar for both the disciplined as well as the undisciplined group (cf. Section 3.2). Second, our analysis did not reveal any peculiarities regarding the performance of these three participants.

4. Analysis and Interpretation

In this section, we present the analysis of our experiment and interpret the results. First, we discuss the results of the statistical analysis. Then, we relate the results to the research question(s) and discuss peculiarities that we detected during the analysis. We discuss task T7 separately, because it diverges in its design from the other tasks (cf. Section 3.5).

4.1 Analysis of Correctness & Response Time

For the analysis of our data, we differentiate between two aspects, according to our research questions: Correctness of solutions and response time for each task. Additionally, we manually analyzed the log data that we collected during the experiment. We used SPSS[7] and R[8] for analysis.

4.1.1 Correctness

For correctness, we used a 3-point scale. A solution could either be completely correct (2), almost correct (1), or wrong (0). We decided to use a 3-point scale, because we found that subjects often solved a task almost correct, such that we can be sure that comprehension has taken place correctly. To this end, we analyzed the log data to filter out minor mistakes that do not depend on the discipline of annotations. For example, in task T1, subjects should count the number of #ifdef variables and one participant used the PROPHET search facility only for one (of two) folders. Hence, she detected only 15 instead of 17 #ifdef annotations (i.e., the corresponding preprocessor variable). Nevertheless, we can be sure that the participant worked seriously on the task, but only made a mistake in using the tool. Besides this example, there are similar cases for the other tasks. Additionally, we defined a threshold for the answers that we found to be almost correct, so that we can guarantee that an answer is (almost) correct. As a result, we decided that only if at least 90% of the task has been solved correctly, we assess the answer with "almost correct".

Furthermore, two authors double-checked their evaluation of the results as follows: Each of the two authors has been assigned a group. First, they checked all tasks of this groups participants, respectively. Afterwards, they hand out their assessment each other. Then, they checked the solutions of the participants and the assessment of the other author. In the case that their assessment for a certain task diverged, this task was discussed together and another author should have been asked. However, for all tasks of both groups, both authors agreed on the assessment of each other, respectively.

Table 4. Overview of correctness for each task

Task	Type of annotation	0	1	2	χ^2	p
Task 1	Disciplined	5	4	2	1.679	0.432
	Undisciplined	5	3	0		
Task 2	Disciplined	1	1	9	1.082	0.598
	Undisciplined	2	1	5		
Task 3	Disciplined	6	0	5	0.038	0.845
	Undisciplined	4	0	4		
Task 4	Disciplined	7	2	2	1.360	0.507
	Undisciplined	3	2	3		
Task 5	Disciplined	4	4	3	1.337	0.512
	Undisciplined	2	5	1		
Task 6	Disciplined	3	4	4	2.796	0.247
	Undisciplined	1	6	1		
Task 7	Disciplined	9	0	2	0.130	0.719
	Undisciplined	6	0	2		

0 – wrong, 1 – almost correct, 2 – completely correct

In Table 4, we give an overview of the correctness of the answers of the subjects. Task T2 seems to be easiest, because it was solved correctly by most subjects. In contrast, task T7 seems to be rather difficult, because there are only a few correct answers, which may be caused by the different design of the task. Overall, the distribution of correct answers is similar in both groups. We conducted a χ^2 test to evaluate whether significant differences between the correctness of tasks exist. Although having a small number of participants, χ^2 is an appropriate test, because researchers have shown that χ^2 is even applicable (and robust) to low expected values [4]. Our test revealed no significant difference between the disciplined and undisciplined group: the χ^2 values are smaller than 2.796 and the p values are larger than 0.247. Hence, we found *no significant evidence* that the kind of annotation has an effect on correctness.

4.1.2 Response Time

In Table 5, we give an overview of the mean response times for each task and group. To analyze the effect of the kind of annotation on response time, we conducted a t test for independent samples [1], because the data have a metric scale and are normally distributed (as shown by a Kolmogorov-Smirnov test [1]). We did not correct the response times for wrong answers, which we discuss in Section 5. Our data reveal that, for the tasks T1 to T6, the difference in the response time is negligible (according to a t test; the t values vary from 1.197 to 1.952; all p values are larger than 0.068). In contrast, the difference between response times for the last task T7 is significant (t value: -3.239; p value < 0.05), with the disciplined group being faster. However, we have to take into account that the task for both groups was different. Hence, the response time should not be compared with those from task T1 to T6.

[7] http://www.ibm.com/software/analytics/spss/

[8] http://www.r-project.org/

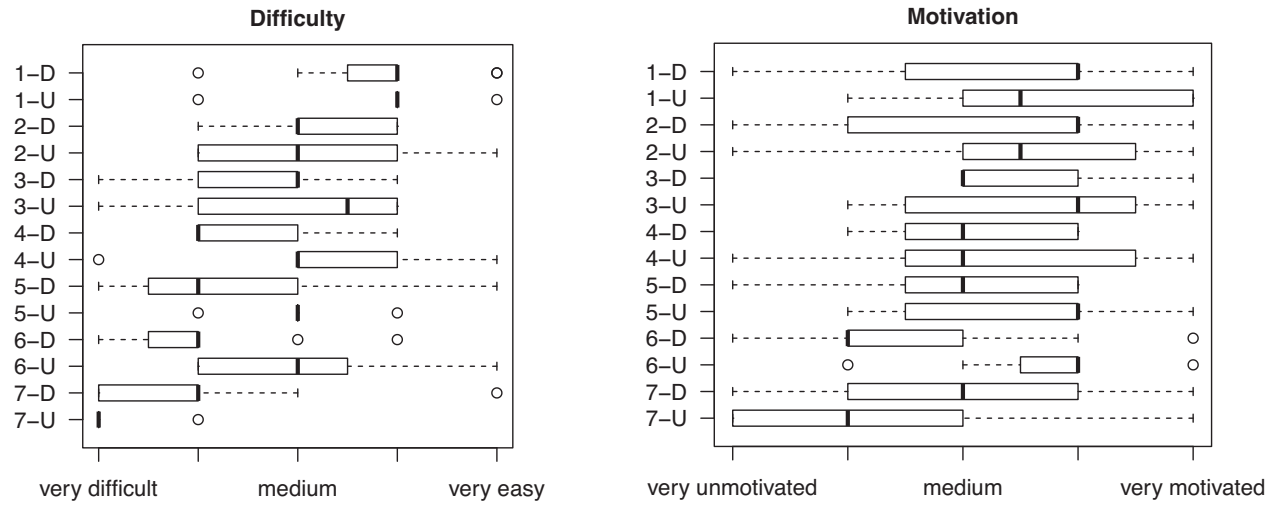


Figure 5. Difficulty (left) and motivation (right) of the tasks (assessed by participants): D – discipined group, U – undisciplined group

4.2 Motivation and Difficulty

For all tasks, we asked the participants to assess the difficulty of the tasks and their motivation to solve them on a five-point Likert scale. The reason for gathering these measures is to eliminate the possibility that neither motivation nor difficulty (of the tasks) may bias our results. We show the results in Figure 5. Overall, the results coincide with those we measured for correctness and response time: There is *no* significant difference between the disciplined and undisciplined group for tasks T1 to T6, and thus there is no influence of these factors on our results. However, the results of task T6 vary compared to the other five tasks in that the motivation of both groups differs considerably, which we cannot explain entirely.

Table 5. Overview of response time for each task

Task	Version	Response time Distribution	Mean	N	t value
Task 1	Discip		10.69	11	1.504
	Undiscip		8.21	8	–
Task 2	Discip		10.59	11	1.197
	Undiscip		9.57	8	–
Task 3	Discip		4	11	1.526
	Undiscip		3.07	8	–
Task 4	Discip		12.44	11	1.698
	Undiscip		9.98	8	–
Task 5	Discip		8.13	11	1.952
	Undiscip		6.46	8	–
Task 6	Discip		11.96	11	1.745
	Undiscip		8.61	8	–
Task 7	Discip		11.64	11	-3.239
	Undiscip	0 5 10 20 30	22.29	8	–

N: number of subjects per group; t value: result of t test ($p < 0.05$)

The differences correspond to the result of the experiment in so far as the undisciplined group performed slightly better than the disciplined group (cf. Figure 4), which may explain our observation to some extent.

Finally, there is an observable difference in motivation and difficulty for the last task T7, which corresponds to the results of our analysis, specifically regarding response time.

4.3 Research Questions

The analysis of our experimental data revealed no significant differences between the disciplined and the undisciplined group, neither for correctness nor for response time. Next, we interpret our results with respect to the research questions we formulated in Section 3.1. Additionally, we put emphasis on error handling, which mainly encompasses our results of task T7. Finally, we present findings that result from a detailed analysis of the log data we recorded during the experiment.

RQ.1 – Does the discipline of annotations affect the correctness of program comprehension and maintenance tasks?

Based on the results of our analysis, we conclude that the discipline of annotations has *no* significant influence on mental simulation or maintenance tasks. Nevertheless, we observed some minor tendencies regarding correctness. First, we observed a considerable difference between the first four tasks, regardless of the respective group. For instance, the second task has been solved correctly[9] by most participants in both groups (disciplined: $\sim$ 90%, undisciplined: 75%), as we show in Table 4. In contrast, for the other three tasks (T1, T3, and T4), the ratio of correct answers is 60% or less for both groups (cf. Table 4). Additionally, we observed the tendency that participants of the undisciplined group performed slightly better with respect to correctness than participants of the disciplined group regarding maintenance tasks ($\sim$ 80% compared to $\sim$ 70%, on average) (cf. Table 4).

Second, while the disciplined group performed slightly better for the first four tasks (i.e., mental simulation tasks), the undisciplined group achieved better results for tasks T5 and T6 (i.e., maintenance tasks). This observation is reflected in the number of wrong answers (relatively to all participants for each of the two groups, see Table 4) and may indicate that the compressed representation of variability by means of undisciplined annotations may be advantageous for making changes to the source code. However, this

[9] either completely or almost correct

is only a conjecture and not supported by our statistical analysis. More research with a specific focus on certain tasks such as maintenance is necessary to evaluate this assumption. Specifically, it is of interest to evaluate whether this observation holds for large-scale maintenance tasks, where the comprehensibility of a program may outweigh the compressed presentation.

RQ.2 – Does the discipline of annotations influence the time needed to solve mental simulation and maintenance tasks?

Similar to the correctness of the answers, our statistical analysis reveals that there is *no* significant difference regarding response time, either. Nevertheless, the mean time for each task shows that the disciplined group tends to need more time throughout all tasks, without being significant. For some tasks, such as T3 or T4, the increased code size could be responsible, because this leads to more code that has to be investigated by the participant. However, we have currently no general explanation for this observation. Overall, we conclude that the kind of annotation does not seem to affect the response time of subjects at all.

Detecting/Fixing errors in the presence of `#ifdefs`

Because T7 differs from all other tasks in both design and focus of the task the results are difficult to compare to the other tasks. Hence, we discuss the results of this task separately. Generally, we made two interesting observations when considering the results of this task: First, *detecting* an error in the presence of preprocessor annotations is a difficult and time-consuming task. This observation is reflected by the high response time of the undisciplined group that is significantly higher than the time of the disciplined group. Second, even with the knowledge that an error exists, it is complicated to *remove* this error in the presence of preprocessor annotations. This is reflected by the high number of wrong answers, even for the disciplined group (where we provided information that an error exists). Although these observations confirm the assumption of other researchers [6], we support this assumption for the first time by means of our experiment.

Nevertheless, both observations have a limited generalizability. First, detecting and correcting errors is a complicated and time-consuming task in general, even without preprocessor annotations. Hence, our results may be influenced by this fact, and we cannot entirely conclude to what extent the preprocessor annotations are the reason for our observations.

4.4 Log Data/Manual Analysis

To get deeper insights into *how* subjects solved the tasks, we analyzed the behavior of subjects during each task. The reason is that we wanted to investigate whether certain patterns occur when solving tasks. As a result, we can reason about wrong answers or exceeding response time and how both are related to the discipline of preprocessor annotations. To this end, we analyzed the comprehensive log data (recorded by PROPHET) and the edited source files (for T5 and T6). Next, we present our observations.

Task T1 to T4 (Mental Simulation): For the first four tasks, the log data reveal that *all* participants had an idea of how to solve each task. For instance, all of them used similar and appropriate search terms. Additionally, we observed a similar scroll behavior, such as time spent on certain code fragments while investigating the source code. Nevertheless, in particular cases (and independent of the respective group), some participants failed to solve the task at all, by proposing a solution that was entirely wrong, or they submitted only a partial or even no solution. We hypothesize that other reasons are responsible for this diverging results, such as time spent on a certain task. But this observation could also indicate that identifying relevant code fragments is generally complicated in the presence of preprocessor annotations and complex tasks. Overall, our log data support the analysis result that the discipline of annotations does not influence correctness.

Task T5 and T6 (Maintenance): For the two tasks related to maintenance, our analysis revealed that four participants had no idea how to solve the tasks. For T5, three participants of the disciplined group provided a solution that was not even close to correct. In fact, they made changes to code that actually had nothing to do with the problem, as stated in the task description. Interestingly, two of these participants did not use the code-search facilities of PROPHET to identify the right place within the code, which could be a reason for their wrong solution. Furthermore, we observed that most of the participants who provided a solution that was partially correct made the same mistakes. That is, while they introduced a new preprocessor variable in the right place, they neglected to remove certain statements or fields from the code that is surrounded by this newly introduced variable. This, in turn, may lead to errors, and thus we decided to classify these solutions as only partially correct. Finally, two participants of the undisciplined group introduced syntax errors, which we assume were caused by the undisciplined nature of annotations. More precisely, the participants omitted and misplaced a bracket, respectively.

For task T6, we identified one participant who failed to remove the source code related to the preprocessor variable `IP_V6` (as specified in the task). Since she used the code search with the same search terms as the other participants, we hypothesize that she did not understand the task at all and thus used a wrong search term.

Overall, we could not detect a clear pattern for one of the groups. However, regarding the minor mistakes for T5, we assume that preprocessor annotations in general (i.e., independent of the discipline) have a (negative) effect on source code changes.

Task T7 (Detecting/Fixing Errors): Finally, for task T7, we have to distinguish between the disciplined and undisciplined group, because they had to solve considerably different tasks. For the disciplined group, the log data revealed that five (out of eleven) participants tried to fix the error stated in the task description at a totally different position in the source code than expected. Hence, we assume that, although we provided information to localize the root of the error, half of the participants did not understand the interrelation between the preprocessor annotations and the syntax error caused by them.

In conclusion, when considering the detailed behavior of subjects, we could not find an influence of the kind of annotation on program comprehension. Nevertheless, we explicitly mention that this does not necessarily mean that there are no (significant) differences. In fact, our conclusion is valid for our experiment, but may not be generalizable regarding further studies. Beyond that, for different tasks and independent of the discipline, our observations indicate that the presence of preprocessor annotations in general has a (negative) influence on program comprehension. Further research on this topic using screen and video-capture facilities could provide deeper insights.

5. Threats to Validity

Next, we discuss threats to internal and external validity, which helps other researchers to interpret our results and to put them into relation to experiments with similar focus. Internal validity refers to how well we controlled influences on what we observed, that is, program comprehension. External validity describes the generalizability of our results [32].

5.1 Internal Validity

The participants had to work in an unfamiliar environment (i.e., PROPHET), specifically designed to support experimental studies.

Still, we argue that this environment is easy enough to use, because it contains standard features of a modern IDE, such as syntax highlighting. Moreover, with the help of PROPHET, we can rule out that other factors, such as outstanding knowledge about tools or techniques (e.g., the preferred IDE or regular expressions), bias our results.

Furthermore, there are some limitations regarding the execution and analysis of our study. Three students did not complete the questionnaire prior to the experiment, which we used to assign participants to the groups. Hence, we randomly assigned these students to the groups. However, both groups are comparable, as the similar experience ranks show that we computed for each group.

Additionally, we did not filter out wrong answers when analyzing the response time, because this would reduce our already small sample size further [36]. However, by manual inspection of our log data, we found no information indicating that a participant did not answer a task seriously (e.g., response times did not deviate too much toward zero).

5.2 External Validity

First, all participants of our experiment are undergraduate students and thus have less programming experience than professional developers. Hence, our results are only valid for this level of programming experience and should only be carefully interpreted with respect to experienced developers. Nevertheless, previous studies demonstrate that even students can be treated similar to professional programmers [35].

Second, participants had to complete the tasks on code snippets of different systems, whereas in a real-world scenario, programmers work on large-scale systems that consist of thousands of lines of code. In addition, the particular tasks were rather small, so that they fit the time constraints of the experiments. Both, the amount of source code and complexity of tasks may limit the generalizability of our study, because they do not reflect the real world in its entirety. However, regarding the tasks, we decided to define *micro tasks* to measure different aspects of preprocessor annotations and program comprehension. The effect of the kind of annotation in larger tasks has to be evaluated empirically, for which our experimental design can be reused.

Third, we created all disciplined annotations manually, by transforming undisciplined ones. This may render the disciplined code artificial and thus limit the generalizability of our case study. However, the disciplined annotations that result from our transformation coincides with those, typically found in C systems [26]. Hence, we argue that creating the disciplined annotations does not affect our case study.

Finally, in our study, we considered only the CPP usage in C programs, while the CPP is used with other languages such as C++ as well. However, all tasks and code examples have been chosen without making heavy use of underlying language mechanisms (i.e., standard imperative mechanisms). Hence, the tasks could be applied to programs in different target languages in the same way.

6. Related Work

Prior to this paper, several other researchers addressed the usage of preprocessor annotations in source code.

Spencer and Collyer investigated the usage of preprocessor annotations to support the portability of systems [34]. They found that a moderate usage of #ifdefs is acceptable, whereas an overly extensive usage leads to severe problems regarding maintenance and understanding of source code. However, compared to our work, they solely rely on experiences with the C News system (and how to avoid unnecessary #ifdefs), whereas we conducted an empirical experiment. Furthermore, they do not distinguish between disciplined and undisciplined annotations.

Feigenspan et al. addressed the problem of comprehensibility of preprocessor annotations [9, 11, 12]. They conducted experiments to measure whether background colors are useful to support program comprehension in the presence of preprocessor annotations. Similarly, Le et al. propose a prototype that provides facilities to manage software variation within a GUI [25]. They present a user study to evaluate differences between using common CPP directives and their prototype, confirming that the prototype is more effective for implementing variability. While both approaches focus on comprehension of annotated code in general (including possible alternatives to CPP), we focus on program comprehension of *different types* of annotations (disciplined and undisciplined), which are often discussed in the literature.

Medeiros et al. analyzed preprocessor-based systems with respect to syntax errors [29]. They conducted experiments on 40 systems and observed that only few errors occur, which particularly remained for years in the system. While this coincides with our observations that syntax errors are hard to detect, our work is different in that we conducted an experiment to determine the favorable annotation discipline, using humans.

Furthermore, in prior work, we analyzed the discipline of annotations with respect to code replication [31]. In particular, we investigated whether the discipline of annotations has an effect on the number of code clones. Within our analysis, we found evidence that systems with entirely disciplined annotations contain more code clones than systems with undisciplined annotations. However, we neither considered program comprehension nor maintenance issues of the analyzed system.

7. Conclusion and Future Work

The C preprocessor CPP is widely used to express variability in source code. Despite its expressiveness and usage even in large-scale systems, the CPP is criticized for obfuscating source code, making it difficult to understand. We concentrated on the issue of how the discipline of preprocessor annotations influences program comprehension, by means of a controlled experiment with human subjects. We created two groups, each of which had to solve seven tasks related to maintenance and mental simulation on source code with disciplined and undisciplined annotations, respectively. Then, we measured their performance in terms of correctness and response time. Our results indicate that the discipline of annotations has *no influence on program comprehension and maintenance*, neither for correctness nor for performance (in terms of response time). Although we observed some tendencies, they are not supported by our statistical analysis. However, our experiment confirms that finding errors in the presence of preprocessor annotations is a tedious and time-consuming task. More research on this topic is needed, especially with a focus on certain aspects that were out of scope of this study such as large-scale maintenance tasks or error detection in the presence or absence of preprocessor annotations. In future work, we aim at addressing these aspects based on the results of this study as a starting point.

First, we plan an experiment to evaluate whether the current results hold for large-scale maintenance tasks. In such an experiment, participants have to solve one task concerned with maintenance, which is more complex than the tasks in the present experiment. Second, we will use a complete system rather than small parts of different systems. In a similar way (e.g., similar experimental setup), an experiment for measuring program comprehension in the presence of different types of annotations is part of our future work.

Acknowledgment

Siegmund's work is funded by BMBF project 01IM10002B. Apel's work is funded by the DFG grants AP 206/2, AP 206/4, and AP 206/5. Schulze would like to thank Bram Adams for initial discussion on that topic during PASED summer school 2011. Finally, we are grateful to Christoph Steup for support in acquiring participants for the experiment and comments on earlier versions of this paper.

References

[1] T. Anderson and J. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[2] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. Work. Conf. Reverse Engineering (WCRE)*, pages 86–95. IEEE, 1995.

[3] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Work. Conf. Reverse Engineering (WCRE)*, pages 281–290. IEEE, 2001.

[4] G. Camilli and K. D. Hopkins. Applicability of Chi-square to 2×2 Contingency Tables with Small Expected Cell Frequencies. *Psychological Bulletin*, 85(1):163, 1978.

[5] A. Dunsmore and M. Roper. A Comparative Evaluation of Program Comprehension Measures. *Journal Sys. and Soft. (JSS)*, 52(3):121–129, 2000.

[6] M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Software Engineering (TSE)*, 28(12):1146–1170, 2002.

[7] J.-M. Favre. The CPP Paradox. In *Proc. European Workshop Software Maintenance*, 1995. `http://equipes-lig.imag.fr/adele/Les.Publications/intConferences/EWSM91995Fav.pdf`.

[8] J.-M. Favre. Understanding-In-The-Large. In *Int. Workshop Program Comprehension (IWPC)*, pages 29–38. IEEE, 1997.

[9] 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*, pages 1–47, 2012.

[10] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring Programming Experience. In *Proc. Int. Conf. Program Comprehension (ICPC)*, pages 73–82. IEEE, 2012.

[11] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proc. Int. Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75. Institution of Engineering and Technology, 2011.

[12] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, M. Frisch, and G. Saake. Supporting Program Comprehension in Large Preprocessor-Based Software Product Lines. *IET Software*, 6(6):488–501, 2012.

[13] J. Feigenspan, N. Siegmund, A. Hasselberg, and M. Köppen. PROPHET: Tool Infrastructure to Support Program Comprehension Experiments. In *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*, 2011. Poster.

[14] A. Garrido and R. Johnson. Challenges of Refactoring C Programs. In *Proc. Int. Workshop Principles of Software Evolution (IWPSE)*, pages 6–14. ACM, 2002.

[15] A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, pages 323–326. IEEE, 2003.

[16] A. Garrido and R. Johnson. Analyzing Multiple Configurations of a C Program. In *Proc. Int. Conf. Software Maintenance (ICSM)*, pages 379–388. IEEE, 2005.

[17] N. Göde and J. Harder. Clone Stability. In *Proc. European Conf. Software Maintenance and Reengineering (CSMR)*, pages 65–74. IEEE, 2011.

[18] C. Goodwin. *Research in Psychology: Methods and Design*. Wiley Publishing, Inc., second edition, 1999.

[19] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.

[20] E. Jürgens, F. Deissenböck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 485–495. IEEE, 2009.

[21] C. Kapser and M. W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *Proc. Work. Conf. Reverse Engineering (WCRE)*, pages 19–28. IEEE, 2006.

[22] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 311–320. ACM, 2008.

[23] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int. Conf. Objects, Models, Components, Patterns (TOOLS)*, pages 174–194. Springer, 2009.

[24] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 492–501. ACM, 2006.

[25] D. Le, E. Walkingshaw, and M. Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Proc. IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150. IEEE, 2011.

[26] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[27] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proc. Int. Conf. Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011.

[28] R. Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 140:1–55, 1932.

[29] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating Preprocessor-Based Syntax Errors. In *Proc. Int. Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2013. to appear.

[30] C. Roy and J. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, Queen's University at Kingston, 2007.

[31] S. Schulze, E. Jürgens, and J. Feigenspan. Analyzing the Effect of Preprocessor Annotations on Code Clones. In *Proc. Work. Conf. Source Code Analysis and Manipulation (SCAM)*, pages 115–124. IEEE, 2011.

[32] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin Company, 2002.

[33] J. Siegmund. *Framework for Measuring Program Comprehension*. PhD thesis, University of Magdeburg, 2012.

[34] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *Proc. USENIX Technical Conf.*, pages 185–197. USENIX Association Berkeley, 1992.

[35] M. Svahnberg, A. Aurum, and C. Wohlin. Using Students as Subjects – An Empirical Evaluation. In *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*, pages 288–290. ACM, 2008.

[36] J. Yellott. Correction for Fast Guessing and the Speed Accuracy Tradeoff in Choice Reaction Time. *Journal of Mathematical Psychology*, 8:159–199, 1971.