The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter

Romero Malaquias*, Márcio Ribeiro†, Rodrigo Bonifácio‡, Eduardo Monteiro§, Flávio Medeiros¶, Alessandro Garcia‖, Rohit Gheyi**

*†Computing Institute, Federal University of Alagoas, Maceió - AL, Brazil
‡Department of Computer Science, §Department of Statistics, University of Brasília, Brasília - DF, Brazil
¶Rio Largo Campus, Federal Institute of Alagoas, Rio Largo - AL, Brazil
‖Informatics Department, PUC-Rio, Rio de Janeiro - RJ, Brazil
**Department of Computing Systems, Federal University of Campina Grande, Campina Grande - PB, Brazil

rbsm@ic.ufal.br, marcio@ic.ufal.br, rbonifacio@cic.unb.br, edumonteiro@unb.br,
flaviomedeiros@ifal.edu.br, afgarcia@inf.puc-rio.br, rohit@dsc.ufcg.edu.br

***Abstract*—The C preprocessor is a simple, effective, and language-independent tool. Developers use the preprocessor in practice to deal with portability and variability issues. Despite the widespread usage, the C preprocessor suffers from severe criticism, such as negative effects on code understandability and maintainability. In particular, these problems may get worse when using undisciplined annotations, i.e., when a preprocessor directive encompasses only parts of C syntactical units. Nevertheless, despite the criticism and guidelines found in systems like Linux to avoid undisciplined annotations, the results of a previous controlled experiment indicated that the discipline of annotations has no influence on program comprehension and maintenance. To better understand whether developers care about the discipline of preprocessor-based annotations and whether they can really influence on maintenance tasks, in this paper we conduct a mixed-method research involving two studies. In the first one, we identify undisciplined annotations in 110 open-source C/C++ systems of different domains, sizes, and popularity GitHub metrics. We then refactor the identified undisciplined annotations to make them disciplined. Right away, we submit pull requests with our code changes. Our results show that almost two thirds of our pull requests have been accepted and are now merged. In the second study, we conduct a controlled experiment. We have several differences with respect to the aforementioned one, such as blocking of cofounding effects and more replicas. We have evidences that maintaining undisciplined annotations is more time consuming and error prone, representing a different result when compared to the previous experiment. Overall, we conclude that undisciplined annotations should not be neglected.**

I. Introduction

The C preprocessor is a widely used tool in practice to deal with portability and variability issues in configurable systems [1]–[3]. To use the preprocessor, developers annotate the variable code by using directives like **`#ifdef`**, **`#elif`**, **`#else`**, and **`#endif`**. Existing studies [4] classified preprocessor-based annotations in disciplined and undisciplined. Undisciplined annotations happen when the preprocessor directive does not encompass the entire subtree in the corresponding abstract syntax tree. In this sense, annotating only the expression of an **`if`** statement (and leaving its body unannotated) represents an undisciplined annotation example.

In a previous study [5], researchers conducted a controlled experiment to test the assumption that *the discipline of annotations has influence on comprehension and maintenance of annotated code.* However, their results do not support this assumption, i.e., they concluded that the discipline has *no* influence on such tasks. Their findings are based on measurements in terms of response time to conclude maintenance and comprehension tasks; and correctness of such tasks. Nevertheless, undisciplined annotations have been strongly criticized, since they make the code harder to read, understand, and maintain [2], [6], [7]. In addition, there are guidelines that ask developers to not use them. For instance, there is a specific recommendation for contributing to the Linux Kernel that states the following: "*Prefer to compile out entire functions, rather than portions of functions or portions of expressions.*"[1]

Given this scenario, we decided to better evaluate whether developers really care about the discipline of preprocessor-based annotations and whether these annotations can really make a difference on software maintenance tasks. Our research is inspired by previous works that contact actual developers using source code repositories (e.g., GitHub) [3], [8] and by the aforementioned experiment [5]. Understanding the advantages and disadvantages of having disciplined annotations is important to guide the development of new tools and improve programming practices on this front.

This way, to better understand whether the discipline matters, we conduct a *mixed-method research* that consists of two studies. In the first one, we focus on the following research question: *Do developers accept suggestions to remove undisciplined annotations?* To answer this question, we initially find several undisciplined annotations in 110 open-source C/C++ systems of different domains, sizes, and popularity GitHub metrics. Then, we manually refactor the code to make the annotations disciplined. We refactor only one annotation per project. Right away, we submit a pull request with

[1]https://tinyurl.com/linuxguidelines

our code changes. We also submit comments and questions using GitHub to understand the developers thoughts about our pull requests. This study extends the one we executed previously [9]. To bring more evidences, here we present four times more pull requests. Also, in this paper we categorize the developers answers by using a word cloud to better understand the reasons behind accepting and rejecting the pull requests.

In the second study, we intend to answer the following research questions: *Does the use of undisciplined annotations increase the time to perform maintenance tasks?* and *Does the use of undisciplined annotations increase the number of errors committed by developers during maintenance tasks?* To answer these questions, we conduct a controlled experiment with 64 participants (undergraduate students). However, while we focus on maintenance tasks to test the same dependent variables (response time and correctness), we have differences when compared to the previous experiment [5]. First, we designed the experiment to (a) block two sources of confounding effects (the experience and involvement of the participants and the set of tasks they should perform) and (b) increase the number of replicas, which allows us to *obtain a more precise estimate of the factor/interaction effect* [10]. Second, the previous experiment [5] used clones to discipline some annotations, making the number of lines of code sometimes very different when compared to the task with undisciplined annotations. In contrast, the number of lines of code of our experiment is fairly comparable, i.e., the disciplined and undisciplined code used in all tasks have almost the same number of lines of code. In addition, in our experiment we do not compute time of "almost correct" tasks. Only entirely complete tasks are considered. We discard incomplete ones. Moreover, we do not make comparisons like correcting an error in disciplined code against identifying an error in undisciplined code. To the best of our knowledge, these tasks are different and should not be comparable. On the other hand, our experiment only has three kinds of maintenance tasks: fix a syntax error, introduce a new feature variant, and fix a semantic error in a specific variant. The previous experiment has seven and includes the three kinds of tasks we consider.

Our results suggest that it is important to take the discipline of annotations into account. Regarding the pull requests we submitted, 99 out of 110 have been answered and decided. Almost two thirds have been accepted. From the set of rejected ones, some developers mentioned they could accept in case we had submitted the pull request to a different code snippet, e.g., not to a deprecated one. In this case, we reach a total acceptance rate of 71%. With respect to the controlled experiment, our observations lead to evidences that maintaining undisciplined code is more time-consuming and error prone. This result contrasts with the previous experiment [5].

In summary, this paper provides the following contributions:

- An empirical study based on pull requests to better understand the thoughts of practioners with respect to the discipline of preprocessor-based annotations. We extend our previous study [9] by (i) presenting four times more pull requests; (ii) asking whether developers would accept the pull requests in case we had submitted to different code snippets; and (iii) categorizing the answers in terms of a word cloud to better reason about the developers thoughts. (Section III);
- A controlled experiment that investigates the effect of the discipline of preprocessor-based annotations in terms of response time and correctness in maintenance tasks. (Section IV).

II. Disciplined vs. Undisciplined Annotations

The C preprocessor is a simple, effective, and language-independent tool. Developers widely use it in practice to deal with portability and variability issues. It is essentially used in all projects written in C and C++, including many well-known web servers, databases, and operating systems [3].

However, the C preprocessor has received strong criticism. Previous studies raise problems regarding the lack of separation of concerns [2], [6], [7], proneness to introduce subtle errors (such as syntax errors, memory leaks, undeclared identifiers etc) [11]–[15], and obfuscation of the source code, making the task of reading and understanding the code more difficult [1], [16]. In particular, these problems may get worse when using undisciplined annotations [3]. In this case, the preprocessor directives encompass only parts of C syntactical units. In this paper, we use the same definition of a previous work [5]: "*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.*" By using this definition, an example of disciplined annotation is an **`#ifdef`** encompassing an entire **`if`** statement. On the other hand, annotating just part of the conditional expression of an **`if`** statement represents an example of undisciplined annotation.

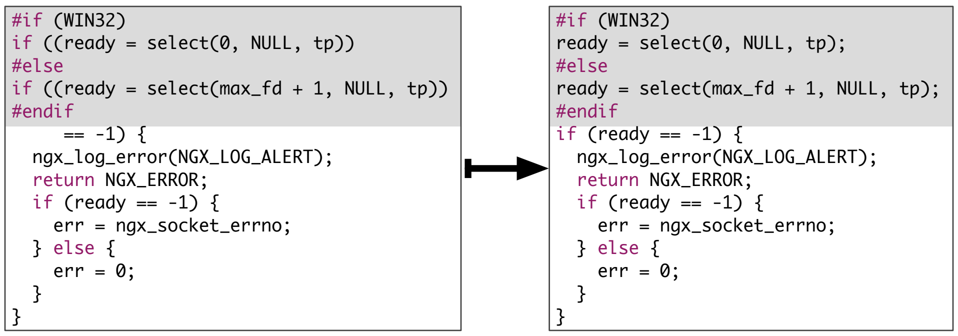
Figure 1 illustrates a code snippet from the *Nginx* web server.[2] The left-hand side contains undisciplined annotations. One of the *Nginx* developers disciplined the annotations. Notice that the annotations at right-hand side contain fragments of code that belong to entire subtrees, i.e., complete assignments.

Despite all criticism regarding undisciplined annotations (not to mention guidelines to avoid them, e.g., in Linux), previous research concluded that the discipline of annotations has no observable effect on comprehension of annotated code [5]. Because undisciplined annotations seem more difficult to read and understand [2], [3], [6], [7], in this paper we intend to better understand the developers thoughts on this front and whether disciplined annotations really *do not matter* on maintenance tasks. To do so, we conduct two studies. The first one focuses on contacting actual developers by using pull requests (Section III); and the second one presents a controlled experiment inspired by the aforementioned one [5] (Section IV).

III. Study 1: Pull Requests

The first study we report in this paper regards the submission of a set of pull requests to discipline existing

[2]http://nginx.org



```
#if (WIN32)
if ((ready = select(0, NULL, tp))
#else
if ((ready = select(max_fd + 1, NULL, tp))
#endif
     == -1) {
  ngx_log_error(NGX_LOG_ALERT);
  return NGX_ERROR;
  if (ready == -1) {
    err = ngx_socket_errno;
  } else {
    err = 0;
  }
}
```

```
#if (WIN32)
ready = select(0, NULL, tp);
#else
ready = select(max_fd + 1, NULL, tp);
#endif
if (ready == -1) {
  ngx_log_error(NGX_LOG_ALERT);
  return NGX_ERROR;
  if (ready == -1) {
    err = ngx_socket_errno;
  } else {
    err = 0;
  }
}
```

Fig. 1. Nginx developer avoiding undisciplined annotations in ngx_select_module.c. Commit: 8292037f7c233f8351b4baf90c84656550cb12cf

preprocessor-based annotations of open-source systems. Our assumption here is that the acceptance rate of the pull requests might serve as an indication that this kind of refactoring is relevant, and thus the differences between disciplined and undisciplined annotations are significant for software maintenance and program comprehension. Now, we present the settings of this study. Then, we present and discuss the results.

A. Settings

We use the pull requests to answer the following question: *Do developers accept suggestions to remove undisciplined annotations?* Answering this question is important to check whether developers feel more comfortable with disciplined annotations. Also, their comments related to the pull requests can bring new perspectives on the benefits and possible side-effects of disciplining preprocessor-based annotations.

To change the code and perform the pull requests, we first need background on how to discipline preprocessor-based annotations. In this context, previous studies propose transformations to turn undisciplined annotations into disciplined ones. However, some of the existing transformations lead to *code clones* [4], [17], [18]—which might be considered an undesirable result. In contrast, in this paper we use a catalog of refactorings to remove undisciplined annotations that does not clone code [19]. We used three refactorings available in the catalog: Refactoring 2, Refactoring 4, and Refactoring 6 (R2, R4, and R6). We used these refactorings because it is common to find opportunities to apply them in practice [9].

Figure 2 illustrates the strategy we performed. For each system, we first use a tool to find undisciplined annotations (Step 1). This tool is able to find opportunities to apply the catalog of refactorings, such as the left-hand side of Figure 1. Because we execute the tool considering the entire systems source code, we can find several undisciplined annotations. Yet, for each project, we randomly select *only one* undisciplined annotation to work with. This is important to minimize bias, such as having several decisions (accepting or rejecting the pull requests) from the same developer. Then, given the selected undisciplined annotation, we apply a proper refactoring from the catalog to make it disciplined and submit a pull request right away (Step 2). Notice that we neither fix bugs nor submit new test cases. We only focus on submitting a disciplined version of the annotation. Upon the request of systems' developers, we submit comments (using GitHub) to better clarify the changes (Step 3). This is a typical approach to have pull requests accepted in open-source systems.

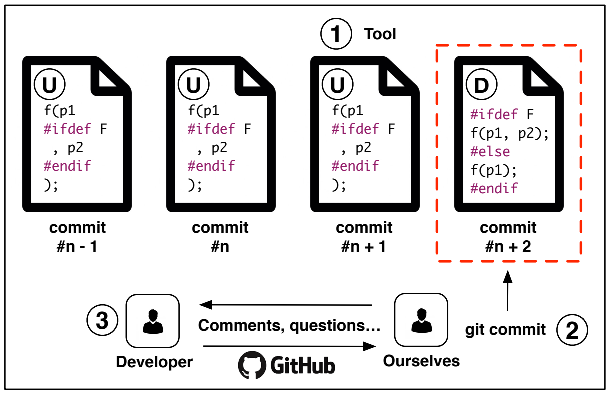


Fig. 2. Our strategy to make suggestions to remove undisciplined annotations.

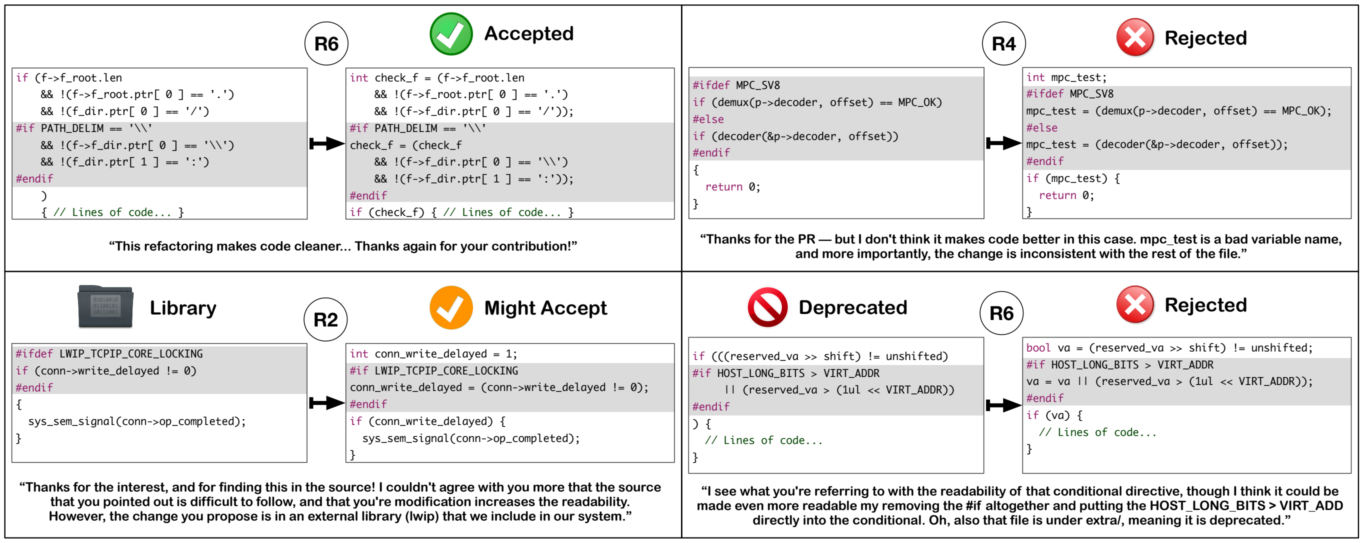
We submit pull requests to several systems written in C/C++ of different domains, sizes, and GitHub popularity metrics, such as *openvpn*, *libpng*, *libxml2*, *cherokee*, *sonyxperiadev*, *SQLite3*, *Arduino*, and *TcpDump*.

B. Results and Discussion

We submitted 110 pull requests, which means we considered 110 preprocessor-based systems. The list of all systems to which we submitted pull requests is available at our companion website.[3] Figure 3 illustrates four pull requests we submitted and the refactorings we used for them (i.e., R2, R4, and R6). At the top of the figure, on the left-hand side, we show a pull request that has been accepted. On the right-hand side, we have a pull request that has been rejected. Figure 3 also shows the developers comments regarding each pull request.

The accepted category is straightforward: it means that the developer accepted the pull request and merged the changes

[3]https://sites.google.com/site/icpc2017easylab/



```
if (f->f_root.len
    && !(f->f_root.ptr[ 0 ] == '.')
    && !(f_dir.ptr[ 0 ] == '/')
#if PATH_DELIM == '\\'
    && !(f->f_dir.ptr[ 0 ] == '\\')
    && !(f_dir.ptr[ 1 ] == ':')
#endif
    )
    { // Lines of code... }
```

```
int check_f = (f->f_root.len
    && !(f->f_root.ptr[ 0 ] == '.')
    && !(f->f_dir.ptr[ 0 ] == '/'));
#if PATH_DELIM == '\\'
check_f = (check_f
    && !(f->f_dir.ptr[ 0 ] == '\\')
    && !(f->f_dir.ptr[ 1 ] == ':'));
#endif
if (check_f) { // Lines of code... }
```

"This refactoring makes code cleaner... Thanks again for your contribution!"

```
#ifdef MPC_SV8
if (demux(p->decoder, offset) == MPC_OK)
#else
if (decoder(&p->decoder, offset))
#endif
{
  return 0;
}
```

```
int mpc_test;
#ifdef MPC_SV8
mpc_test = (demux(p->decoder, offset) == MPC_OK);
#else
mpc_test = (decoder(&p->decoder, offset));
#endif
if (mpc_test) {
  return 0;
}
```

"Thanks for the PR — but I don't think it makes code better in this case. mpc_test is a bad variable name, and more importantly, the change is inconsistent with the rest of the file."

```
#ifdef LWIP_TCPIP_CORE_LOCKING
if (conn->write_delayed != 0)
#endif
{
  sys_sem_signal(conn->op_completed);
}
```

```
int conn_write_delayed = 1;
#if LWIP_TCPIP_CORE_LOCKING
conn_write_delayed = (conn->write_delayed != 0);
#endif
if (conn_write_delayed) {
  sys_sem_signal(conn->op_completed);
}
```

"Thanks for the interest, and for finding this in the source! I couldn't agree with you more that the source that you pointed out is difficult to follow, and that you're modification increases the readability. However, the change you propose is in an external library (lwip) that we include in our system."

```
if (((reserved_va >> shift) != unshifted)
#if HOST_LONG_BITS > VIRT_ADDR
     || (reserved_va > (1ul << VIRT_ADDR))
#endif
) {
  // Lines of code...
}
```

```
bool va = (reserved_va >> shift) != unshifted;
#if HOST_LONG_BITS > VIRT_ADDR
va = va || (reserved_va > (1ul << VIRT_ADDR));
#endif
if (va) {
  // Lines of code...
}
```

"I see what you're referring to with the readability of that conditional directive, though I think it could be made even more readable my removing the #if altogether and putting the HOST_LONG_BITS > VIRT_ADD directly into the conditional. Oh, also that file is under extra/, meaning it is deprecated."

Fig. 3. Examples of pull requests we submitted. Below each transformation, we illustrate the messages we got from the developers.

into the system repository. To better analyze the rejected pull requests, we consider two additional categories besides the rejected itself: *deprecated* and *third-party code*. The deprecated category is also straightforward. It means we submitted a pull request to deprecated code. So, it makes no sense to accept it. Third-party code means we submitted a pull request to a code written by third parties (e.g., libraries) that is somehow mixed with the core system code. In this context, almost all developers we interacted do not allow contributions to such code. Instead, some of them suggested to perform the pull request directly to the third-parties repositories. Despite rejecting, notice that the motivations behind these rejections are not necessarily related to our changes, but to *where* we performed them. At the bottom of Figure 3, we illustrate one pull request for each of these two categories. We present the developers comments as well.

The barplot of Figure 4 illustrates the acceptance rate of our pull requests. So far, 11 pull requests out of 110 had no decision. The plot of Figure 4 disregards them. So, we present the results based on 99 pull requests. Notice that, when considering the ones we have a decision, almost two thirds (63%, 62 pull requests) have been accepted. As mentioned, we did not implement any additional test to check our pull requests. Nevertheless, developers agreed to accept the changes regardless of having new test cases. In fact, no pull request has been rejected exclusively because of tests absence. Still on the accepted pull requests, 43 (69%) out of 62 have been accepted with no changes needed. On the other hand, 19 (31%) required some changes, such as to follow code guidelines specific of each system (e.g., variable names, blank spaces, indentation of **`#ifdef`** directives etc). Now we show some comments of three developers that accepted submitted pull requests:

> "*That's much better.*"
> "*Easier to read.*"
> "*Ok, thanks for the fix. Further improvements are welcome.*"

Another developer mentioned that refactoring undisciplined annotations appears even in to do lists. "*I agree. In fact, it's in the libpng17 TODO list: Refactor preprocessor conditionals to compile entire statements.*" Now we present comments of three developers that rejected the pull requests:

> "*We generally don't do stylistic-only changes.*"
> "*Not only is the code you wrote wrong, I also think it's not needed and probably even harder to understand.*"
> "*I'd call this code harder to read.*"

Notice that we have positive and negative comments regarding code comprehension. We find this scenario not surprising since we are dealing with personal opinions. However, the majority believes that our changes based on disciplined annotations improve program comprehension. Some developers, on the other hand, clearly consider that disciplining the annotations is just a matter of code style. In this context, they say they have lots of bugs and more serious problems to deal with and rather should focus on them. Thus, they rejected our pull requests. Still on the style, we got some other rejections because the disciplined "style" we introduced is not replicated throughout the rest of the code. In addition, since we are not experts in the code we submitted the pull requests, we unintentionally introduced some bugs in a very few scenarios. For obvious reasons, these pull requests were rejected. To have a general view of the answers, we fit all of them in 16 categories we defined. Figure 5 presents the answers according to these categories in terms of a word cloud.

We also investigate the rejected pull requests we submitted to deprecated and third-parties code. We asked the developers whether they would accept these pull requests if it was not

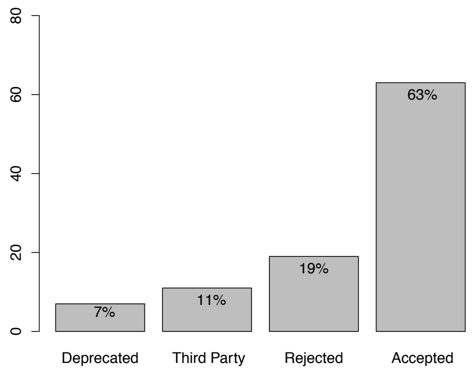


Fig. 4. Results of our pull requests.

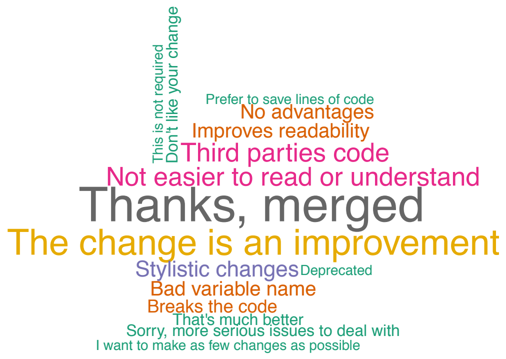


Fig. 5. Word cloud presenting 16 categories of answers.

for the fact we submitted to deprecated and third-parties code. In this sense, 9 (50%) out of 18 developers answered they would accept. If we consider these cases as potential accepts, we reach a total acceptance rate of 71%. In what follows, we present two positive answers (might accept) and one negative (reject anyway):

> "*Sure, but I would love to see them on the core project.*"
> "*Oh yes, I would say it is a better implementation.*"
> "*Sorry, we have to be practical here - so many more serious bugs and issues.*"

We now present the results with respect to each refactoring we used from the catalog [19] (see Table I). Notice that Refactoring 2 was applied just in a few pull requests. When considering the "Might accept" cases as accepts, Refactoring 4 and Refactoring 6 reach 74% and 64% of acceptance rate, respectively. We notice a slightly better tendency of acceptance in favor of Refactoring 4, but we need further studies to deliver significant conclusions.

Based on our results, almost two thirds of the developers accepted our suggestions to remove undisciplined annotations. Overall, these results suggest that undisciplined annotations are important and should not be neglected.

TABLE I
DISTRIBUTION OF THE PULL REQUESTS DECISIONS PER REFACTORING (R2, R4, AND R6 [19]).

Refactoring	Status	Occurrences
2	Accepted	3
	Might accept	1
	Rejected	1
4	Accepted	40
	Might accept	7
	Rejected	16
6	Accept	19
	Might accept	1
	Rejected	11

C. Threats to Validity

Submitting one pull request per system threats to external validity. Our decision on this front however was to avoid two or more answers from the same developer. Yet, we believe we minimize this threat because we have a great number of pull requests in a great variety of systems domains and sizes. In addition, the small number of refactorings we used from the catalog (Refactoring 2, Refactoring 4, and Refactoring 6) to discipline annotations [19] represents a threat to external validity as well. We used three refactorings (see Figure 3). In this sense, we cannot generalize our results to other refactorings that discipline preprocessor-based annotations. However, we used these three refactorings due to the high frequency of opportunities to apply them in practice [9]. In other words, the left-hand sides of Figure 3 are common in existing code of open-source systems. In addition, all refactorings we used focuses on **`if`** statements, again representing a threat to external validity. However, notice that the three refactorings can also be applied to other statements (e.g., **`while`**, **`for`** etc) [19].

Regarding the rejected pull requests we submitted to deprecated and third-parties code, 9 (50%) out of 18 developers answered they would accept in case we had submitted the pull requests to the core project. In the total rate acceptance (71%), we included these cases. This decision represents a threat since these "acceptances" were not merged.

In some cases, developers asked why they should accept our suggestions to remove the undisciplined annotations. We then argued in favor of disciplined annotations, e.g., mentioning that code understandability could be improved. In this sense, our comments could induce developers to accept some pull requests and thus represent a threat. However, we minimize this threat because this is the situation that indeed happens in real practice: in general, when developers submit pull requests, they believe their submissions improve the existing code and make comments in favor of the them.

IV. Study 2: Controlled Experiment

In this section we present our controlled experiment. Here we mitigate two limitations of [5]: the lack of replication and blocking. We present the settings, experimental units, and then discuss the results.

A. Study Settings

Our investigation aims to understand the effect of undisciplined annotations on software maintenance tasks. To do so, we focus on the following two research questions: *Does the use of undisciplined annotations increase the time to perform maintenance tasks?* and *Does the use of undisciplined annotations increase the number of errors committed by developers during maintenance tasks?* We answer our questions by comparing undisciplined annotations with disciplined ones. To answer both research questions, we measure the time necessary for a given subject to conclude a set of maintenance tasks; and the errors he committed during these tasks (trials). Accordingly, we state the following null hypotheses:

- $H1_0$: there is no significant difference in the time necessary to maintain disciplined (the control treatment) and undisciplined (the treatment under investigation) annotations;
- $H2_0$: there is no significant difference in the number of errors committed by developers when maintaining disciplined (the control treatment) and undisciplined (the treatment under investigation) annotations.

Since we are comparing two treatments, in case we reject the null hypotheses, we only have to compare the mean value of the control and treatment observations to estimate the effect of the undisciplined approach on maintenance tasks.

The design of our experiment blocks two variables: (a) the participant skills and engagement; and (b) two sets of maintenance tasks. The Latin square design of order two with replicas is suitable in this situation, where the goal is to compare two treatments and block two variables [20]. Each Latin square replica comprises two participants (randomly assigned to the rows of the squares) and two sets of maintenance tasks (representing the columns of each square, i.e., ST_1 and ST_2). Figure 6 presents the design of our experiment. We also randomly set the treatments that each participant should use in ST_1 and ST_2. For example, the 4th participant of Figure 6 should solve ST_1 using undisciplined annotations (U) and ST_2 using disciplined ones (D) in this order. Since we have several participants taking part of the experiment, our design consists of several Latin squares of order two.

This design uses randomization to assign the participants to the squares and assign the treatments to the cells of each square. Each treatment appears once in each row and column. This way, we block the two sources of variability: the participants and the two sets of tasks. The design also leads to one replica for each Latin square (increasing the number of errors' degrees of freedom) [21].

B. Experimental Units

The participants of our experiment are 64 undergraduate students from three different courses at the University of Brasília, Brazil: Computer Science, Computer Engineering, and Mechatronic. All students were enrolled in an advanced course on programming techniques and have 3 to 5 semesters of experience in programming (particularly using imperative languages such as C, C++, and Java). They also have some previous experience using the C preprocessor. The participants were randomly organized in an initial set of 32 Latin squares and received a 50 minutes training on the subject of configurable systems using the C preprocessor. Our training material had examples of both disciplined and undisciplined annotations, but we did not mention these terms. We only focused on showing two different ways of annotating preprocessor-based code. Before the actual experiment, we asked the participants to perform warmup tasks comprising disciplined and undisciplined annotations.

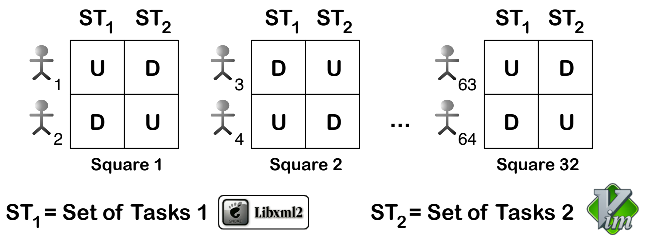


Fig. 6. Design of the experiment using Latin squares.

For the experiment execution, we have prepared two virtual machines, each one having six pre-configured instances of the Eclipse IDE. We have one instance of the Eclipse for each distinct maintenance task. Figure 7 illustrates our setup. Virtual Machine 1 should be used by participants that solve ST_1 (comprising $T1$, $T2$, and $T3$) using disciplined annotations and ST_2 (comprising $T4$, $T5$, and $T6$) using undisciplined annotations in this order. In Virtual Machine 2, we invert our treatments, according to the Latin square design.

Our tasks are based on existing code of two highly-configurable systems, LIBXML2 and VIM, which have been used in several other studies [4], [5], [11], [16]. We adapt and simplify all tasks to fit the scope of our research and to allow the participants to conclude them in a section of at most 150 minutes. We have three kinds of tasks for both systems (see Figure 8). In the first one, participants should fix a syntax error by identifying and removing an additional parenthesis in one specific configuration (variant). In the second one, there are alternative function calls annotated with `#ifdef` and `#else` directives. The task consists of adding a new variant (using `#elif`, for example). In addition, some parameters (described in the task) should be annotated with directives as well. Last but not least, in the third one, the participants should fix a semantic error. Several parts of an arithmetic expression are annotated with preprocessor directives. One variant computes an incorrect value. We indicate the problematic variant and expect the participants to fix it. Altogether, we have prepared 12 maintenance scenarios: six with disciplined annotations and six with undisciplined ones. To simplify the distribution of the participants for the experiment execution, we configured two laboratories of the Computer Science Department of the University of Brasília, Brazil (see "*Laboratory 1*" and

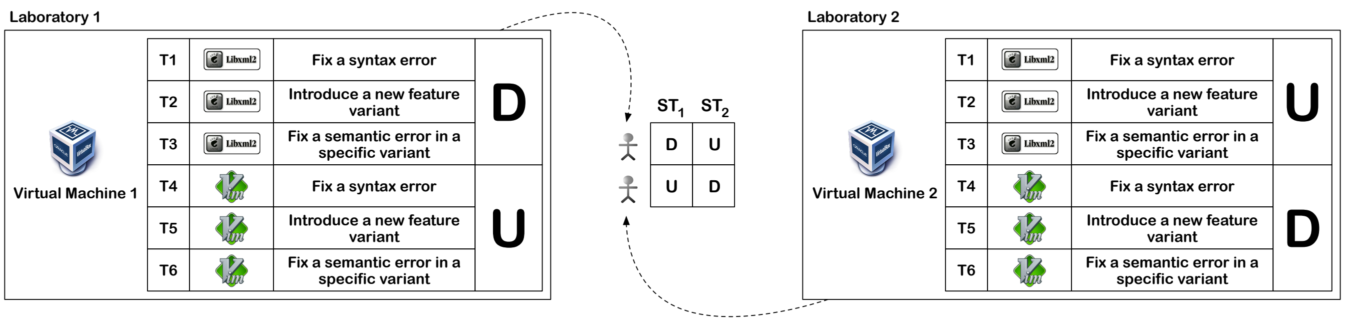

Laboratory 1
Virtual Machine 1
T1
Libxml2
Fix a syntax error
T2
Introduce a new feature variant
T3
Fix a semantic error in a specific variant
T4
Fix a syntax error
T5
Introduce a new feature variant
T6
Fix a semantic error in a specific variant
D
U
ST1
ST2
D
U
U
D
Laboratory 2
Virtual Machine 2
T1
Fix a syntax error
T2
Introduce a new feature variant
T3
Fix a semantic error in a specific variant
T4
Fix a syntax error
T5
Introduce a new feature variant
T6
Fix a semantic error in a specific variant
U
D

Fig. 7. Structure of the experiment in terms of experimental units.

T1: Fix a syntax error

```c
int tst = c[0] != ':';
#ifdef SUPPORT_IP6
tst = (tst && c[3] != ' ');
#if defined PF_INET
tst = tst && (c[1] != '/' ||
              c[4] != ' ');
#else
tst = tst && (c[2] != '0' ||
              c[4] != ' ');
#endif
#else
tst = (tst && c[3] != '-'));
#endif
if (tst) { ... }
```

```c
if (c[0] != ':' && (
#ifdef SUPPORT_IP6
c[3] != ' '
#if defined PF_INET || AF_INET
&&
#if defined PF_INET
(c[1] != '/'
#else
(c[2] != '0'
#endif
|| c[4] != ' ')
#endif
#else
c[3] != '-')
#endif
)) { ... }
```

T2: Introduce a new variant

```c
#ifdef HAVE_SP
l = sp(buf, "PORT %d,%d,%d,%d,%d,%d",
       a[0] & 0xff, a[1] & 0xff,
       a[2] & 0xff, a[3] & 0xff,
       p[0] & 0xff, p[1] & 0xff);
#else /* HAVE_SNP */
l = snp(buf, sizeof(buf),
       "PORT %d,%d,%d,%d,%d,%d",
       a[0] & 0xff, a[1] & 0xff,
       a[2] & 0xff, a[3] & 0xff,
       p[0] & 0xff, p[1] & 0xff);
#endif /* HAVE_SNP */
```

```c
#ifdef HAVE_SP
l = sp(buf, "PORT %d,%d,%d,%d,%d,%d",
#else /* HAVE_SNP */
l = snp(buf, sizeof(buf),
       "PORT %d,%d,%d,%d,%d,%d",
#endif /* HAVE_SNP */
       a[0] & 0xff, a[1] & 0xff,
       a[2] & 0xff, a[3] & 0xff,
       p[0] & 0xff, p[1] & 0xff);
```

T3: Fix a semantic error in a specific variant

```c
#else
r = 5;
#ifdef PF_INET
#ifdef AF_INET
r = r * ( 10 - xml(22, 30, 32)
          + prt) * pl;
#else
r = r * ( 10 - xml(22, 30, 32)
          - px + pl);
#endif
#else
#ifdef AF_INET
r = r * ( 10 - xml(22, 31, 37)
          + prt) * pl;
#else
r = r * ( 10 - xml(22, 31, 37)
          - px + pl);
#endif
#endif
#endif
```

```c
#else
r = 5 * ( 10 - xml(22,
#ifdef PF_INET
30, 32)
#ifdef AF_INET
+ prt) * pl)
#else
- px + pl)
#endif
#else
31, 37)
#ifdef AF_INET
+ prt) * pl)
#else
- px + pl)
#endif
#endif
#endif
;
```

Fig. 8. Code snippets of the LIBXML2 tasks. We have three analogous ones for VIM. The snippets are simplified due to space restrictions.

"*Laboratory 2*" at the top of Figure 7).[4]

C. Execution and Data Analysis Procedures

We first randomly assign the 64 participants to the 32 Latin square replicas. Then, we randomly assign the treatments (Disciplined $\times$ Undisciplined) to the cells of each Latin square. Actually, after assigning a treatment to the first cell of a Latin square of order two, the remaining cells are automatically defined within that particular square.

We then conducted the participants to either *Laboratory 1* or *Laboratory 2*, depending on their assignments. After all virtual machines were set, we detailed the tasks and the procedures they should follow. That is, the participants should begin by executing the Eclipse instance related to $T1$. Then, they should start a chronometer (*Play* button) using a plugin we have implemented for the experiment, and perform the maintenance task. Our plugin has also a *Pause* button, for asking questions during the experiment, for example; and a *Finish* button, when done. When the solution is submitted (by pressing *Finish*), the plugin automatically runs a set of test cases to check whether the participant indeed concluded the task. In case of success, we stop and record the time. Then, such participant is allowed to close the Eclipse instance regarding $T1$ and proceed to $T2$ (using the corresponding Eclipse instance). Similar instructions should be performed in $T2$—$T6$. If a test fails, we increase the number of errors (trials) counter and let the participant continue until success is accomplished.

Perhaps due to the nature of the tasks, only 30 participants concluded all tasks within 150 minutes. Differently from the previous experiment [5], we do not consider "almost correct" results. We decided to adopt a conservative approach, disregarding the data (time and number of errors) of any participant that did not accomplish successfully all tasks on time. This way, some of our squares will be incomplete (with only one row). Therefore, to maximize the number of replicas, we randomly rearranged the participants that concluded all tasks in new Latin squares. When considering only this final set of observations (30 participants), we also found several time observations below 4 minutes. This value is close to the first quartile, i.e., 3 minutes and 46 seconds. To avoid a possible threat related to time measurements of participants that did not start the chronometer in the expected moment (*Play* button), we decided to use a linear regression method as imputation strategy for the time observations below 4 minutes. Again, this is a conservative approach we follow to avoid bias. In summary, from a total of 180 observations (six tasks, 30 participants), we imputed 27 observations based on tasks with disciplined annotations and 21 observations based on

[4]The artifacts that we used in our experiment are available at the companion website: https://sites.google.com/site/icpc2017easylab/

undisciplined ones. After the data imputation, we proceeded with an exploratory data assessment and tested the hypotheses introduced in Section IV-A using the Analysis of Variance method (ANOVA), as we detail in the next section. Before the hypothesis testing, we checked whether the response data satisfies the ANOVA assumptions.

D. Results and Discussion

The boxplot of Figure 9 shows some descriptive statistics related to the total time spent by the participants to conclude all tasks. Some relevant information could be drawn from this figure. For instance, the total time median when using undisciplined annotations (63 minutes) is almost 90% greater than the median when using disciplined annotations (35 minutes). Most of the observations related to the total time to conclude all tasks, when using disciplined annotations, lies between 30.00 and 52.75 minutes (the first and third quartiles, respectively); in contrast, most of the observations when using undisciplined annotations lies between 49.00 and 72.50 minutes. There is almost no overlapping between the two "boxes," which leads to some evidences that performing maintenance tasks with undisciplined annotations is more *time consuming*.

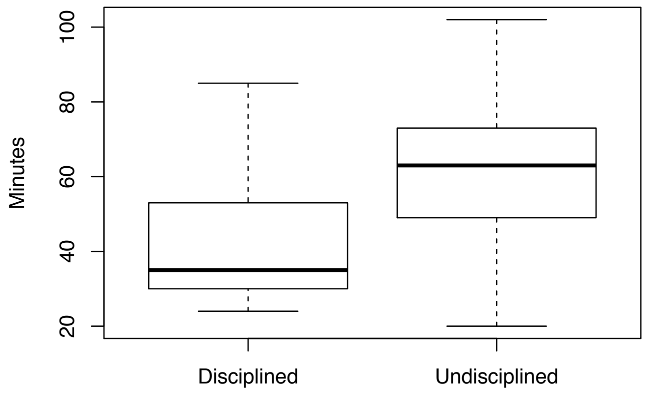


Fig. 9. Total time to conclude all tasks.

Figure 10 shows a density plot of the differences $Tu - Td$, that is, the time that each participant spent using undisciplined annotations (Tu) and the time that each participant spent using disciplined annotations (Td). Notice that most participants spent more time when using undisciplined annotations, and thus most of the differences correspond to positive values. However, some participants—seven (23%) out of 30—were able to conclude all tasks using undisciplined annotations more quickly than when using disciplined ones. The median of the time differences for those cases is 17 minutes. However, one participant spent 43 minutes less solving the tasks when using undisciplined annotations than when using disciplined ones.

Besides the exploratory data assessment, we test the first hypothesis (*H1*) of this study using the Analysis of Variance (ANOVA) technique. We first validate the model using the *Global Validation of Linear Model Assumptions* [22], which is a global test for the typical assumptions that might constraint

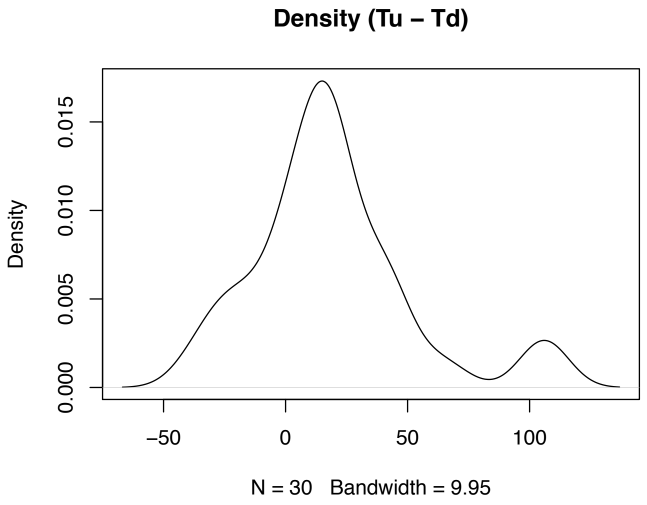


Fig. 10. Density plot considering time differences $Tu - Td$, where Tu is the time a participant spent using undisciplined annotations and Td is the time a participant spent using disciplined ones.

the use of ANOVA: Linearity, Homoscedasticity, Uncorrelatedness, and Normality. To test these assumptions, we used the GVLMA R library and found that the model is acceptable. We then tested our first null hypothesis and found evidences for rejecting it (p$-value = 0.004 < 0.05 = \alpha$). This leads to the first conclusion of this second study.

Based on both exploratory data analysis and hypothesis testing, we conclude that the use of undisciplined annotations increases the time when performing maintenance tasks in preprocessor-based systems.

We follow a similar approach to investigate the second hypothesis, which relates to *error-proneness* when maintaining preprocessor-based systems. We first carry out an exploratory data analysis. Figure 11 shows a boxplot that presents some descriptive statistics related to the number of errors committed by the participants (i.e., number of trials necessary to conclude all tasks). Notice that the number of trials median when using undisciplined annotations (15.0 trials) is almost 300% greater than the number of trials median when using disciplined annotations (4.50 trials). However, differently from the boxplot of Figure 9, Figure 11 presents a moderate overlapping. Also, notice that most of the observations related to disciplined annotations lies between 3.00 and 10.75 (the first and third quartiles, respectively), whereas most of the observations using undisciplined annotations lies between 6.00 and 23.25. Consequently, we cannot infer a sound conclusion about our second hypothesis by only observing the boxplot of Figure 11.

We found that seven[5] participants (23%) out of 30 submitted more answers when using disciplined annotations before correctly concluding the experiment. The remaining participants submitted more trials when using undisciplined ones.

[5]These seven participants are not necessarily the same seven we cited when discussing the first hypothesis. This number is just a coincidence.

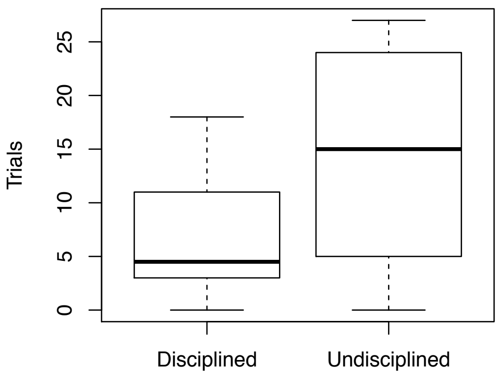


Fig. 11. Number of trials to conclude all tasks.

This suggests that maintaining preprocessor-based systems with undisciplined annotations is more error prone. We also investigate our second hypothesis (*H2*) using ANOVA. The regression model in this case also complies with the ANOVA assumptions. We tested the null hypothesis and also found evidences for rejecting it (with a $\text{p}-value = 0.0001 < 0.05 = \alpha$. This leads to the second conclusion of this second study.

Based on the ANOVA hypothesis testing, we conclude that the use of undisciplined annotations is more error prone when performing maintenance tasks in preprocessor-based systems.

One last interesting result regards tasks $T2$ and $T5$ (introduce a new variant). We observed that eight out of 30 (26%) participants disciplined the annotations (four for $T2$ and four for $T5$) when they received undisciplined ones, even though we did not mention anything on this front during the entire experiment. One participant did the opposite for $T5$.

E. Threats to Validity

In this section we report potential threats to the validity of our experiment. We consider the classification of Wohlin et al. [23]. First, regarding *Conclusion Validity*, notice that our conclusions build upon a high statistical power of regression models that satisfy all ANOVA assumptions—thus we mitigate two possible sources of threats: *low statistical power* and *violated assumptions of statistical tests*. *Reliability of measures* also concerns to *Conclusion Validity*, and a possible threat to our experiment relates to the procedures the participants should have taken to measure the time. Probably, some participants did not start the chronometer at the beginning of their tasks. However, notice that this situation is likely to occur in both treatments, i.e., disciplined and undisciplined annotations. In addition, we decided to impute all suspicious data (time measurements below the first quartile). Despite not presenting the results in the paper, we also carried out a data analysis without performing the imputation, and the results also lead to evidences to reject the first null hypothesis of the experiment. Notice that **the imputation procedure was not needed for the second hypothesis**, since there was no problem when gathering the number of errors data.

The Latin square design deals with sources of *Construct Validity* threats (such as *Mono-operation bias* and *Interaction of different treatments*) [23]. In addition, we considered in our experiment two distinct observations: the time to conclude maintenance tasks (effort) and the number of errors (trials) submitted until concluding the maintenance tasks (error proneness). There is a moderate correlation between these two measurements (0.61 Pearson's correlation), and thus we mitigate another possible source of *Construct Validity*, named *Mono-method bias* [23]. In addition, there is a certain cross-validation of the results of our experiment with the results of the first study reported in this paper, which is based on *Pull Requests*. Also, we tried to avoid most possible sources of bias. For instance, the results when applying the imputation procedure were slightly more favorable to undisciplined annotations (less time to conclude the tasks), even though we report evidences to conclude that this approach is more time consuming during maintenance tasks. Those decisions mitigate possible *Social threats to construct validity*.

There is an extensive debate between Internal x External validity [23], [24]. In summary, if the goal is to establish a cause-effect relationship involving treatments and response variables, the use of rigorous controlled experiments is needed—typically including students to increase the number replicas. This supports the internal validity of the work, though hinders the possibility to generalize the results to different situations. Accordingly, we cannot generalize the results of our experiment to other maintenance scenarios—although we explored three typical maintenance tasks involving preprocessor-based code: fix a syntax error, introduce a new feature variant, and fix a semantic error in a given variant. We cannot generalize our results to the population of professional developers as well, since the participants of our experiment were undergraduate students. However, some studies argue in favor of using students to conduct controlled experiments [25], [26].

Our strategy in this paper was to conduct a *multi-method research* that initially considered the opinion of open-source systems developers with respect to the benefits of *disciplining* preprocessor-based annotations (the first study); and then we performed a controlled experiment to better understand some *causal relationships* of using undisciplined annotations on typical maintenance tasks (the second study). In this situation, the first study favors external validity, while the second study favors internal validity.

V. Implications

Developers introduce undisciplined preprocessor directives even in projects with explicit coding guidelines targeting undisciplined annotations, such as in the Linux Kernel [4]. Thus, these guidelines are not sufficient to prevent developers from introducing undisciplined annotations in the source code. Our results suggest that project managers and senior developers need to enforce guidelines more efficiently to avoid undisciplined annotations, for example, by performing code reviews, by rejecting patches with undisciplined annotations, and by discussing this topic in email lists to make developers aware of the problem. Furthermore, based on our results, it seems important to develop tools to IDEs (such as Eclipse, Emacs, and Sublime) or extend existing ones [4], [9], [17], [27] to detect undisciplined annotations and warn developers.

As mentioned, Schulze et al. [5] performed a controlled experiment regarding the discipline of preprocessor-based annotations and found that the discipline of annotations has no observable effect on code comprehension and maintenance tasks. They mentioned, however, that more research on this topic is needed. In this paper, we claim the same, specially because we achieved different results and because several practitioners accepted our pull requests, demonstrating that they do not neglect undisciplined annotations.

VI. Related Work

Researchers and practitioners have criticized the C preprocessor because of its limited separation of concerns and code obfuscation that make maintenance and code comprehension difficult [4], [6], [7], [17], [28]. In particular, undisciplined preprocessor annotations have been related to error proneness [7], [11], [17], [28], hindered code understanding and maintainability [2], [6], [7], and limitations in tool support [6], [17], [28]–[30]. An empirical study by Liebig et al. [4] revealed that 15.6% of the preprocessor annotations in 40 open source C projects are undisciplined.

As mentioned throughout the paper, like Schulze et al. [5], we focused on the same dependent variables (time and correctness), but found different results. This way, we conclude that more research should be conducted on the discipline topic. Previously [3], we interviewed 40 developers and performed a survey with 202 participants to understand their preferences regarding the discipline of annotations. Most of the developers agreed that undisciplined annotations impact negatively on maintenance and error proneness. By using a different method (pull requests), we achieved similar results.

There are some approaches to refactor C code with preprocessor directives. For instance, Baxter and Mehlich proposed DMS, a source-code transformation tool for C/C++ [2], [6]. The DMS tool focuses on reverse engineering to gather design information and to ease maintenance tasks. Platoff et al. [31] also proposed the PTT tool to refactor C. However, these tools use heuristics and limit developers to annotate only disciplined annotations, that is, conditional directives that surround only entire functions, type definitions, and statements. Other approaches consider undisciplined preprocessor directives. For instance, Garrido and Johnson [17], [27] developed the CRefactory tool, that provides C refactorings such as renaming functions and extracting macros [29]. Liebig et al. [4] also proposed a variability-aware refactoring approach to refactor C with undisciplined annotations, considering all possible configurations of the source code at the same time [28]. Garrido and Johnson [27] and Schulze et al. [5] also discussed about ways of converting undisciplined into disciplined annotations by cloning code. To minimize bias in our experiments, we used refactorings that do not clone code [19].

Bugs that appear only in some configurations have been detected in popular C systems, such as the *Linux Kernel*, *Apache*, and *Libssh*. Many studies have analyzed software repositories to understand the characteristics of bugs that appear only in some configurations [11], [32]–[34]. Abal et al. [32] and Tartler et al. [35] analyzed the *Linux Kernel* repository to study these bugs. In this sense, previous studies show that the use of undisciplined annotations is error prone [7], [11], [17], [28]. To better support these studies, we considered correctness in our experiment by counting how many times the participants submitted incorrect solutions.

VII. Concluding Remarks

In this paper, we conducted two studies with respect to the discipline of preprocessor-based annotations. In the first one, we submitted 110 pull requests to discipline undisciplined annotations we found. Only 11 have no decision yet. Almost two thirds of the pull requests (63%) have been accepted by the developers of the systems. However, because we are dealing with opinions, we got contradicted answers (e.g., improves vs. does not improve code comprehension). On the other hand, 18% of the pull requests got rejected because we submitted them to deprecated and third-parties code. Nevertheless, when asking the developers whether they would accept if it was not the case, 50% responded positively. We then reached a total acceptance rate of 71%. So, most of the developers we contacted care about the discipline of preprocessor-based annotations. Thus, undisciplined annotations seem important and thus should not be neglected.

In the second study, we conducted a controlled experiment to investigate a possible *causal relationship* between maintaining undisciplined preprocessor-based code and two software engineering attributes: productivity and error proneness. Accordingly, we measured and analyzed the total time and the total number of errors observed during the execution of our experiment—involving 30 participants that implemented six typical maintenance scenarios adapted from real systems. As a result, we found evidences that maintaining undisciplined preprocessor-based code is a time-consuming and error-prone task, in comparison with the activity of maintaining code that only uses disciplined annotations.

In summary, when considering the data, setups, systems, and tasks of our work, we have evidences to take the title of this paper and let **`TAG`** be disabled!

VIII. Acknowledgments

This work has been partially supported by CAPES/Procad grant 175956, CAPES/PGCI grant 117875, by CNPq (grants 460883/2014-3, 307190/2015-3, 483425/2013-3, 309884/2012-8, 312149/2016-6, 465614/2014-0, 306610/2013-2), FAPEAL PPGs 14/2016, FAPERJ (grants E26-102.166/2013, E26-225.207/2016), and DEVASSES grant PIRSES-GA-2013-612569.

References

[1] H. Spencer and G. Collyer, "Ifdef considered harmful, or portability experience with C news," in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1992, pp. 185–197.

[2] I. Baxter, "Design maintenance systems," *Communication of the ACM*, vol. 35, no. 4, pp. 73–89, 1992.

[3] 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 29th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl, 2015, pp. 999–1022.

[4] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 191–202.

[5] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter?: A controlled experiment," in *Proceedings of 12th International Conference on Generative Programming: Concepts and Experiences (GPCE)*, 2013, pp. 65–74.

[6] I. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2001, pp. 281–290.

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

[8] K. Narasimhan and C. Reichenbach, "Copy and paste redeemed (T)," in *International Conference on Automated Software Engineering*. IEEE/ACM, 2015, pp. 630–640.

[9] 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*, 2017, To appear.

[10] J. Antony, *Design of experiments for engineers and scientists*. Elsevier, 2014.

[11] F. Medeiros, M. Ribeiro, and R. Gheyi, "Investigating preprocessor-based syntax errors," in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013, pp. 75–84.

[12] C. Kästner and S. Apel, "Virtual separation of concerns – a second chance for preprocessors," *Journal of Object Technology (JOT)*, vol. 8, no. 6, 2009.

[13] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 989–1000.

[14] 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 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2015, pp. 35–44.

[15] L. Braz, R. Gheyi, M. Mongiovi, M. Ribeiro, F. Medeiros, and L. Teixeira, "A change-centric approach to compile configurable systems with #ifdefs," in *Proceedings of 15th International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2016, pp. 109–119.

[16] 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 32nd International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.

[17] A. Garrido and R. Johnson, "Analyzing multiple configurations of a C program," in *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 379–388.

[18] A. Garrido and R. E. Johnson, "Embracing the C preprocessor during refactoring," *Journal of Software: Evolution and Process*, vol. 25, no. 12, pp. 1285–1304, 2013.

[19] F. Medeiros, M. Ribeiro, R. Gheyi, and B. Fonseca, "A catalogue of refactorings to remove incomplete annotations," *Journal of Universal Computer Science*, vol. 20, no. 5, pp. 746–771, 2014.

[20] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: Design, innovation, and discovery*. Wiley-Interscience, 2005.

[21] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[22] E. A. Pena and E. H. Slate, "Global validation of linear model assumptions," *Journal of the American Statistical Association*, vol. 101, no. 473, pp. 341–354, 2006.

[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[24] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 9–19.

[25] D. I. Sjoberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren, and M. Vokác, "Conducting realistic experiments in software engineering," in *Proceedings of the 1st International Symposium on Empirical Software Engineering (ISESE)*. IEEE, 2002, pp. 17–26.

[26] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjectsa comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, pp. 201–214, 2000.

[27] A. Garrido and R. Johnson, "Refactoring C with conditional compilation," in *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*. IEEE, 2003, pp. 323–326.

[28] C. Kästner, P. 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 Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2011, pp. 805–824.

[29] A. Garrido and R. Johnson, "Challenges of refactoring C programs," in *Proceedings of International Workshop on Principles of Software Evolution*. ACM, 2002, pp. 6–14.

[30] Y. Padioleau, "Parsing C/C++ code without pre-processing," in *Compiler Construction*. Springer, 2009, pp. 109–125.

[31] M. Platoff, M. Wagner, and J. Camaratta, "An integrated program representation and toolkit for the maintenance of C programs," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 1991, pp. 129–137.

[32] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of 29th International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2014, pp. 421–432.

[33] B. J. Garvin and M. B. Cohen, "Feature interaction faults revisited: An exploratory study," in *Proceeding of the 22nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2011, pp. 90–99.

[34] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr., "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, 2004.

[35] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90,000 #ifdefs issue," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 421–432.