Trabalho de Conclusão de Curso

An Empirical Study on the Frequency of Disciplined and Undisciplined Annotations in Preprocessor-Based Systems in C and C++

José Carlos Viana Filho
jcvf@ic.ufal.br

Orientadores:

Márcio de Medeiros Ribeiro
Ana Carla Gomes Bibiano

Maceió, September de 2021

José Carlos Viana Filho

An Empirical Study on the Frequency of Disciplined and Undisciplined Annotations in Preprocessor-Based Systems in C and C++

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciências de Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Márcio de Medeiros Ribeiro

Ana Carla Gomes Bibiano

Maceió, September de 2021

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Lívia Silva dos Santos – CRB-4 – 1670

V614e Viana Filho, José Carlos.
An empirical study on the frequency of disciplined and undisciplined annotations in preprocessor –based systems in C and C++ / José Carlos Viana Filho. – 2021.
23 f.:il.

Orientador: Márcio de Medeiros Ribeiro.
Coorientadora: Ana Carla Gomes Bibiano.
Monografia (Trabalho de Conclusão de Curso em Ciência da Computação) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2021.

Bibliografia: f. 22-23

1. C e C++ (linguagens de computador). 2. Software. 3. Refatoração. 4. Anotações Condicionais. I. Título.

CDU: 004.43

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Márcio de Medeiros Ribeiro - Orientador
Universidade Federal de Alagoas

Ana Carla Gomes Bibiano - Coorientador
Pontificia Universidade Catolica do Rio de Janeiro

Baldoino Fonseca dos Santos Neto - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Thiago Damasceno Cordeiro - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, September de 2021

Agradecimentos

Agradeço aos meus pais, meus irmãos, minha esposa (te amo!), meus amigos e conhecidos, meus professores, meu orientador e minha co-orientadora, por conseguir concluir esse trabalho. Tudo o que faço carrega um pouco do eu sou, mas eu sou todos os relacionamentos e interações acumulados ao longo da vida. Obrigado por tudo.

Resumo

As linguagens de programação C e C++ permitem uma ferramenta de preprocessador para escrever anotações condicionais. Comunidades de software relevantes tais como Linux e Apache tem usado anotações condicionais em seus projetos. No entanto, desenvolvedores escrevem estas anotações condicionais de uma forma indisciplinada muitas vezes. A aplicação de anotações condicionais indisciplinadas ou anotações indisciplinadas pode ter um efeito negativo sobre a legibilidade do código e o aumento da propensão de erros. Ao longo desses últimos 10 anos, estudos empíricos tem investigado como o uso de anotações indisciplinada tem afetado a qualidade do software e como disciplinar essas anotações. Uma estratégia proposta para resolver anotações indisciplinadas é aplicar refatorações. Refatoração é uma transformação de código que almeja melhorar a qualidade do código preservando o comportamento do programa. No contexto de anotações disciplinadas, uma refatoração almeja disciplinar uma anotação indisciplinada. No entanto, não existem evidências empíricas sobre até que ponto o número de anotações disciplinadas tem aumentado e/ou diminuído ao longo desses últimos dez anos. Principalmente, se desenvolvedores aplicam refatorações sobre anotações disciplinadas e se essas refatorações disciplinam essas anotações. Baseado nessas limitações da literatura, esse estudo almeja verifica se a frequência de anotações indisciplinadas e disciplinadas ao longo desses últimos dez anos, e se desenvolvedores aplicam refatorações sobre anotações indisciplinadas na prática. Nós investigamos 23 projetos de software que foram investigados dez anos atrás sobre anotações indisciplinadas. Nossos resultados apresentam que somente 10 projetos de sobre tiveram um aumento significativo no número de anotações disciplinadas nesses últimos dez anos, nós também apresentamos 19 refatorações que foram aplicadas na prática. Nós observamos que desenvolvedores refatoram não sobre anotações disciplinadas, mas também anotações indisciplinadas. Esses resultados podem motivar estudos futuros para investigar se contribuições e recomendações de refatorações existentes tem ajudado desenvolvedores para disciplinar anotações ou se os desenvolvedores tem usado o conhecimento empírico existente para resolver anotações disciplinadas.

Palavras-chave: Anotações Condicionais, Anotações Disciplinadas, Anotações Indisciplinadas, Refatoração, Mineração de Repositório de Sistemas de Softwares

Abstract

The C and C++ programming languages allow a preprocessor tool to write conditional annotations. Relevant software communities such as Linux and Apache have used conditional annotations in their software projects. However, developers write these conditional annotations in an undisciplined way. The application of undisciplined conditional annotations or undisciplined annotations can have a negative effect on code readability and increasing the error proneness. Over the last ten years, empirical studies have investigated how the use of undisciplined annotations can affect the software quality and how to discipline these annotations. A proposed strategy to solve undisciplined annotation is to apply refactorings. Refactoring is a code transformation that aims to improve the code quality, preserving the program behavior. In the context of undisciplined annotations, one refactoring aims to discipline it. However, there is no empirical evidence on to what extent the number of disciplined annotations increased and/or decreased over the last ten years. Mainly, if developers applied refactorings on undisciplined annotations and these refactorings disciplined these annotations. Based on those literature limitations, our study aims to verify if the frequency of disciplined and undisciplined annotations during the last ten years, if developers apply refactorings on undisciplined annotations in the practice. We investigated 23 software projects that were investigated ten years ago on undisciplined annotations. Our results presented that only ten software projects had an increase in the number of disciplined annotations over the last ten years, and we presented 19 refactorings that were applied in practice. We observed that developers refactor not only undisciplined annotations but also disciplined ones. These results can motivate future studies to investigate if existing findings and recommendations of refactorings have helped developers to discipline annotations or if developers have used the existing empirical knowledge to solve undisciplined annotations.

Keywords: Conditional Annotations, Disciplined Annotations, Undisciplined Annotations, Refactoring, Mining Software Repositories

List of Figures

3.1 Study Steps 9
3.2 ifdef-catcher algorithm flow 11
3.3 Malaquias Heuristic of Refactoring Candidates 12
4.1 Distribution of changes in the percentage of disciplined annotations 15
4.2 Projects with Refactorings Candidates 16
4.3 Refactors over time 19

Contents

List of Figures iii

1 Introduction 1

2 Background 3
- 2.1 C Preprocessors 3
- 2.2 Disciplined and Undisciplined Annotations 3
- 2.3 Refactoring on Conditional Annotations 4
- 2.4 Related Work and Existing Limitations 6

3 Study Settings 8
- 3.1 Goal, Research Questions 8
- 3.2 Study Steps 9
  - 3.2.1 Select projects 9
  - 3.2.2 Collect annotations 9
  - 3.2.3 Analyze annotations. 10
  - 3.2.4 Analyze refactorings. 11

4 Results 13
- 4.1 Frequency of Disciplined and Undisciplined Annotations along the last Ten years 13
- 4.2 Refactorings on Undisciplined and Disciplined Annotations along the last Ten years 16

5 Threats to Validity 20

6 Conclusion 21

References 22

1

Introduction

Complex systems are difficult to manage and maintain. The C and C++ languages have preprocessor directives or annotations, and these annotations are used to add variability to software development and to handle this complexity Liebig et al. (2011); Fenske et al. (2020). These directives can be used in the form of disciplined and undisciplined annotations Liebig et al. (2011). Disciplined annotations can be defined as annotations that do not break the structure of elements of the AST (Abstract Syntax Tree) Schulze et al. (2013), do not break any structure of the language. Undisciplined annotations, on the other hand, break the elements of the AST. Undisciplined annotations can cause code readability issues and can make it difficult or impossible to use parser tools that handle or manipulate AST Liebig et al. (2011).

Knowing the problems caused by undisciplined annotations, one solution is to write the annotation in disciplined form or discipline undisciplined annotations. Over the last ten years, empirical studies have investigated how to discipline these annotations. A proposed strategy to solve undisciplined annotation is to apply refactorings. Refactoring is a code transformation that aims to improve code quality, preserving the program behavior. In the context of undisciplined annotations, one refactoring aims to discipline it Medeiros et al. (2017). However, the literature is limited about how if developers have disciplined annotations in the practice, and if developers have applied refactorings on undisciplined annotations. These limitations can guide two research questions: to what extent has the number of disciplined annotations increased or decreased over the last ten years? Do developers refactor undisciplined annotations? The answers to these questions are necessary to have minimal empirical evidence if the effort of the existing studies has helped to improve software quality, decreasing the number of undisciplined annotations.

We then performed our study to answer these questions. We investigated a set of 23 repositories investigated on undisciplined annotations ten years ago Liebig et al. (2011). Using a tool called cppstats Liebig et al. (2011), we created a script to extract and analyze some metrics

from these projects. We then collected the frequency of disciplined and undisciplined annotations of these projects. Based on this analysis, we selected projects that had a significant increase of disciplined annotations, because these projects can be considered candidates to have refactorings. We then used a heuristic on these projects that were selected to search the refactoring candidates Malaquias (2018). Therefore, we applied a manual analysis to confirm these refactorings.

The results showed that the proportion of disciplined and undisciplined annotations remained stable over the 10 years, considering all 23 projects. This indicates that there has not been a remarkable evolution or change in this proportion, leading to observe that there have not been many refactorings. We have found that only ten software projects had a significant increase of undisciplined annotations. We then executed the previously mentioned heuristic on these ten projects to collect refactorings. The analysis resulted in 19 refactorings. We performed a manual analysis, and we found that developers refactor not only undisciplined annotations but also disciplined (undisciplined them). These results confirmed that developers refactor annotations, but do not reveal the motivation behind them. Is it something motivated by theory, or some practical necessity? These results can motivate future studies to investigate if existing findings and recommendations of refactorings have helped developers to discipline annotations or if developers have used the existing empirical knowledge to solve undisciplined annotations.

2

Background

2.1 C Preprocessors

The C and C++ programming languages allow a preprocessor tool in which developers can write conditional or optional annotations Liebig et al. (2011); Fenske et al. (2020). These annotations can be controlled by a macro Fenske et al. (2020); Medeiros et al. (2017). That preprocessor tool is a language independent tool that was initially proposed for working on different hardware and operational systems Medeiros et al. (2017). The C preprocessor is often used in many open and industrial software projects from several domains, allowing these systems to customize their demands from conditional annotations. The Linux Kernel project has over 26 million lines of code and over 15 thousand configuration options, the Apache web server, and Hewlett-Packard's printer firmware also use conditional annotations from the C preprocessor tool Fenske et al. (2020). However, that tool does not change since 70's year. Therefore, the lack of evolution and maintenance on these preprocessors has become difficult the improvement of the conditional annotation application. Besides, developers write these conditional annotations of an undisciplined way. The application of undisciplined conditional annotations can have a negative effect on code readability and increasing the error proneness.

2.2 Disciplined and Undisciplined Annotations

.

We adopted the following definition of a disciplined annotation Schulze et al. (2013) *"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."* Therefore, a disciplined annotation is an `#ifdef` encompassing an entire if statement. On another hand, an undisciplined annotation is when annotating just part of the conditional expression of

an if statement. Figure 2.1 presents an example of an undisciplined annotation and Figure 2.2 showed a disciplined annotation. These annotations are from the Emacs software project [1]. This source code was found in the commit 3d1afd119. We can observe in Figure 2.1 the directive is undisciplined because the #ifdef and #endif annotations is inside the conditional expression, and in Figure 2.2 is a disciplined annotation because the the #ifdef annotation is before the conditional expression and the #endif annotation is after the conditional expression.

```
want_this=
#ifdef HAVE_XFT
      (nlen > 6 && strncmp(name, 'Xft', 4) == 0)
      OR strcmp(XSETTINGS_FONT_NAME, name) == 0
      OR
#endif
      strcmp (XSETTINGS_TOOL_BAR_STYLE, name) == 0
```

Listing 2.1: Undisciplined Annotation

```
want_this =  strcmp (XSETTINGS_TOOL_BAR_STYLE, name) == 0
#ifdef HAVE_XFT
      if ((nlen > 6 && strncmp(name, 'Xft', 4) == 0)
      OR strcmp(XSETTINGS_FONT_NAME, name) == 0)
      want_this = true
#endif
```

Listing 2.2: Disciplined Annotation

Existing studies have investigated how the use of undisciplined annotations can affect the software quality and how to discipline these annotations. Mainly, because undisciplined annotations are not aligned with syntactic units Liebig et al. (2011). It degrades the code comprehension Ernst et al. (2002); Le et al. (2011); Lohmann et al. (2006), increasing the code complexity and affecting negatively the code readability Baxter and Mehlich (2001). Moreover, it can increase fault proneness Abal et al. (2014); Ernst et al. (2002); Ferreira et al. (2016), and harm maintainability McCloskey and Brewer (2005). In that way, studies that have investigated how undisciplined and disciplined annotations are applied in practice is very relevant because it can help to improve the existing approaches, helping developers to know how to discipline these annotations and improve the code quality.

2.3 Refactoring on Conditional Annotations

Along the last ten years, researchers defined approaches to help developers to remove undisciplined conditional annotations, one of these approaches is the application of refactor-

[1] https://github.com/emacs-mirror/emacs

ings Medeiros et al. (2017). Refactoring is a code transformation that aims to improve the code quality preserving the program behavior Fowler (1999). In the context of undisciplined annotations, a refactoring aims to discipline it. Improving then the code quality because undisciplined annotations increase the code complex, worse the code readability, degrading the code quality. Medeiros et al. (2017) propose a catalog of refactorings to discipline undisciplined annotations. They classified the refactorings in four categories : *single statements*, *conditions*, *wrappers*, and *comma-separated elements*. These categories are detailed as follows.

- *Single statements.* In this category, language tokens are duplicated to encompass with preprocessor annotations only entire statements. [2] An example of a language that can be duplicated is the `return` token to encompass an entire statement. The Medeiros' catalog presents variations that use fresh local variables to keep condition expressions in situations in which these sub expressions are complex enough to justify the introduction of new variables. Developers, then, can select the variation according to their concerns.

- *Conditions.* This refactoring is to solve undisciplined annotations surrounding Boolean expressions (used in if and while statements). Developers can use a fresh variable to maintain the statement's conditions. In that case, variables can not have the same identifier name in the same scope. In that way, developers might rename the variable(s) to choose a suitable identifier based on the real responsibility of the source code.

- *Wrappers.* This category presents three types of refactorings to wrap C statements. Preprocessor annotations are often used to wrap C statements in different ways, but it can help the application of an undisciplined annotation. First wrapper refactoring can be applied in case of undisciplined preprocessor usage: alternative statements, developers then need a fresh variable to keep the statement condition. Second wrapper refactoring can be used a fresh variable to preserve the statement's condition and to discipline the preprocessor annotations. Third wrapper refactoring is to remove if statements ending with an else statement. In this case, developers can replace the else by another if statement to resolve the undisciplined usage of the preprocessor.

- *Comma-separated elements.* In this refactoring, developers can set a precondition that the original code does not define a macro PARAM or contains a token with that name, such as a type definition or identifier. If they change a macro definition that the original code is already using, it can change the program behavior. In this way, the catalog suggest modifications to the code locally without global impact. Developers might handle other types of comma separated elements, such as array and enum elements, with a similar refactoring.

[2] A single statement contains no compound blocks. Examples of single statements are variable initializations, function calls, and return statements.

The figure 2.2 presented an example of a refactoring applied on an undisciplined annotation. As mentioned, the figure 2.1 was an undisciplined annotation, developer then disciplined this annotation when he/she added all conditional expression inside the `ifdef` and `endif` annotations. Developer also added a boolean value on the `want_this` variable, it preserves the program behavior to this variable to be true if the conditional expression to be true. Thus, this refactoring is the *conditions* category because it was applied to solve a boolean expression, and from this refactoring the annotation was disciplined and the program behavior was kept.

2.4 Related Work and Existing Limitations

Fenske et al. (2020) investigated the comprehension of developers about C preprocessors. They aimed to understand if the objective correctness of developers during the program comprehension is aligned to the subjective preference of a certain style of conditional directives. In this study, 521 developers evaluated original code (undisciplined directives) and refactored code (disciplined code). In their results, they observed that the most developers had a better comprehension on the undisciplined directives, few developers had a better comprehension on the disciplined directives. A previous work Medeiros et al. (2015) interviewed developers about the use of preprocessors, and specifically on the relation between conditional directives and error proneness. Their results presented that developers perceived the use of preprocessors as an elegant solution. However, developers are aware of the problems of the preprocessors application, such as the increasing of error proneness. In this study, developers mentioned that avoid to use preprocessors and when it is necessary, they try to follow the existing guidelines of best practices applying conditional directives. Also, developers reported to fix bugs that are related to the use of directives.

Malaquias et al. (2017) submitted pull requests on undisciplined directives, aiming to evaluate if developers are interested in to discipline these directives. This study also evaluated if disciplined directives improve the software maintenance. The developers accepted the most of the pull requests and the results presented that the use of undisciplined directives is more time-consuming and error-prone, thus degrading the software maintenance. Other study Liebig et al. (2011) described types of disciplined and undisciplined directives according to the practice. They analyzed 40 software projects and presented recommendation to discipline some types of undisciplined directives.

Another study investigated the number of possibilities of refactorings to apply on undisciplined directives Medeiros et al. (2017). They provided a catalog of refactoring types to apply on undisciplined directives. This study provides empirical evidences that developers prefer the use of refactored code (disciplined directives). Developers accepted 75% of patches that were submitted according to the proposed catalog. Besides, the study presented that refactorings of catalog preserved the system behavior. These results increase the confidence of this catalog of

refactorings did not change the system behavior and it has a high acceptance of developers.

However, despite to the preposition that refactorings can discipline annotations, there are no empirical evidences on to what extent the number of disciplined annotations increased and/or decreased over the last ten years. Mainly, if developers applied refactorings on undisciplined annotations and these refactorings disciplined these annotations in the practice.

3

Study Settings

This chapter presents the study goal, research questions and the steps of this study.

3.1 Goal, Research Questions

Liebig et al. (2011) presented a study about the disciplinarity of annotations in 2011. Along the last ten years, researchers defined a set of refactorings to discipline undisciplined annotations (Medeiros et al. (2017)). Other studies evaluated if the refactored code improved the code readability along the software development. However, despite the preposition that refactorings can discipline annotations, there are no empirical evidences on to what extent the number of disciplined annotations increased and/or decreased over the last ten years. Mainly, if developers applied refactorings on undisciplined annotations and these refactorings disciplined these annotations. Based on those literature limitations, the main goal of this study is to verify if the frequency of disciplined and undisciplined annotations during the last ten years. Mainly, the literature about if disciplined annotations have increased due to the application of refactorings on undisciplined annotations in this last ten years.

RQ_1:To what extent the number of disciplined annotations increased/decreased over the last ten years?

Studies have presented how to discipline undisciplined annotations along this last ten years. Based on that, it is expected that the number of disciplined annotations have increased and the number of undisciplined annotations have decreased along this last ten years in practice. We aimed to present if these expectations is a reality or not in practice. We then presented the frequency of disciplined and undisciplined annotations over the last ten years.

RQ_2:Do developers refactor undisciplined annotations?

Previous studies presented refactorings to discipline undisciplined annotations, but the literature is limited to know if developers apply these refactorings in practice. We aimed to present scenarios in which developers applied refactorings on undisciplined and they disciplined these annotations through these refactorings.

3.2 Study Steps

Figure 3.1 presents the steps of our study. First, we selected projects that were investigated in a previous study in 2011, exactly ten years ago, to evaluate to what extent the number of disciplined annotations increased/decreased over the last ten years. Second, we collected refactorings that were applied to undisciplined annotations. Each study step is detailed as follows.

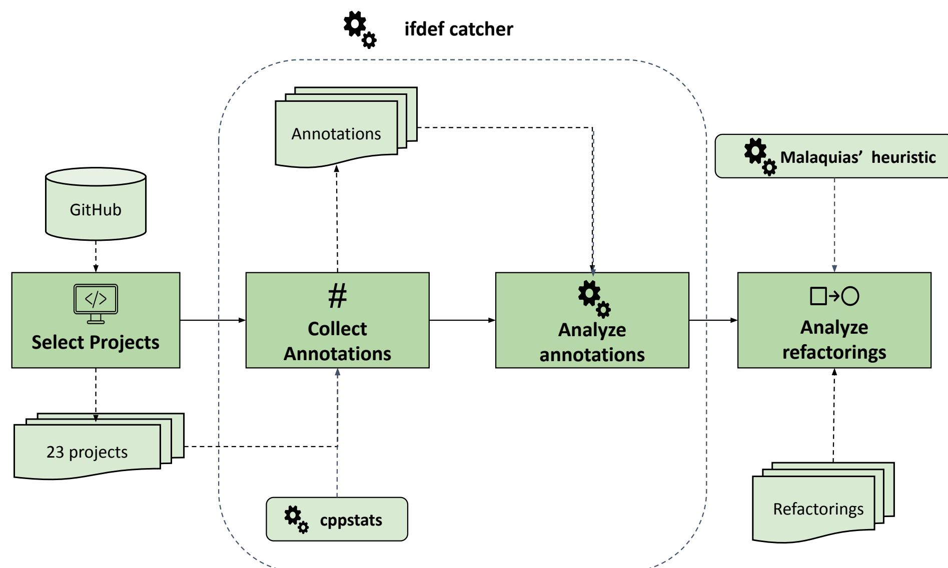


Figure 3.1: Study Steps

3.2.1 Select projects

The projects were selected from the previous study Liebig et al. (2011), which served as a conceptual starting point for our study. However, the source code for some of these projects was not found, which were promptly discarded. The table 3.1 presents the selected projects used in the study.

3.2.2 Collect annotations

We used the Cppstats tool to count and classify the annotations of a source code. This tool was detailed in Liebig et al. (2011). The version of this tool used was the most recent in

Project	Domain	old version	new version
apache	Web server	2.2.11	2.4.46
berkeley.db	Web server	4.7.25	18.1
cherokee	database system	0.99.11	1.2.104
clamav	antivirus program	0.94.2	0.104
cpython	program interpreter	2.6.1	3.9.3
dia	diagramming software	0.96.1	0.97.2
emacs	text editor	22.3	27.1.91
freebsd	operating system	7.1	12.2
gcc	compiler framework	4.3.3	10.2.0
gimp	graphics editor	2.6.4	2.10.24
glibc	programming library	2.9	2.33
gnumeric	spreadsheet application	1.9.5	1.12.49
gnuplot	plotting tool	4.2.5	5.4.1
libxml2	XML library	2.7.3	2.9.10
lighttpd	Web server	1.4.22	1.4.59
linux	operating system	2.6.28.7	5.12.5
openldap	LDAP directory service	2.4.16	2.4.58
parrot	virtual machine	0.9.1	2.9.1
php	program interpreter	5.2.8	8.0.4.rc1
postgres	database system	8.4.beta2	13.2
sqlite	database system	3.6.10	3.35.4
tcl	program interpreter	8.5.7	8.6.11
xterm	terminal emulator	2.4.3	3.1.7c

Table 3.1: Selected projects

their repository[1]; the version at the time of the study did not contain execution instructions and depended on other tools whose versions were not found (Liebig et al. (2011)). We created a script called IfDef Catcher[2] to extract metrics (amount of annotations, percentage of disciplined annotations, and percentage of undisciplined annotations) from the output of the Cppstats tools, using the older and recent versions of each selected project. Figure 3.2 illustrates the flow of the IfDef Catcher. The objective of this script was to collect data, obtaining a general picture of the evolution of the use of disciplined and undisciplined annotations of the selected projects over time. To achieve the goal of evolutionary analysis, we did need to discard unmodified source code over time, or source code with unchanged annotations. An intermediate step of this script was to use a filter that did just that.

3.2.3 Analyze annotations.

After collecting the metrics for each project, tables and graphs were created to identify projects with a marked evolution in the number of annotations. This step allows us to answer our first

[1]https://github.com/clhunsen/cppstats

[2]https://github.com/easy-software-ufal/ifdef-catcher/tree/main/phase-2-quantitative-analysis

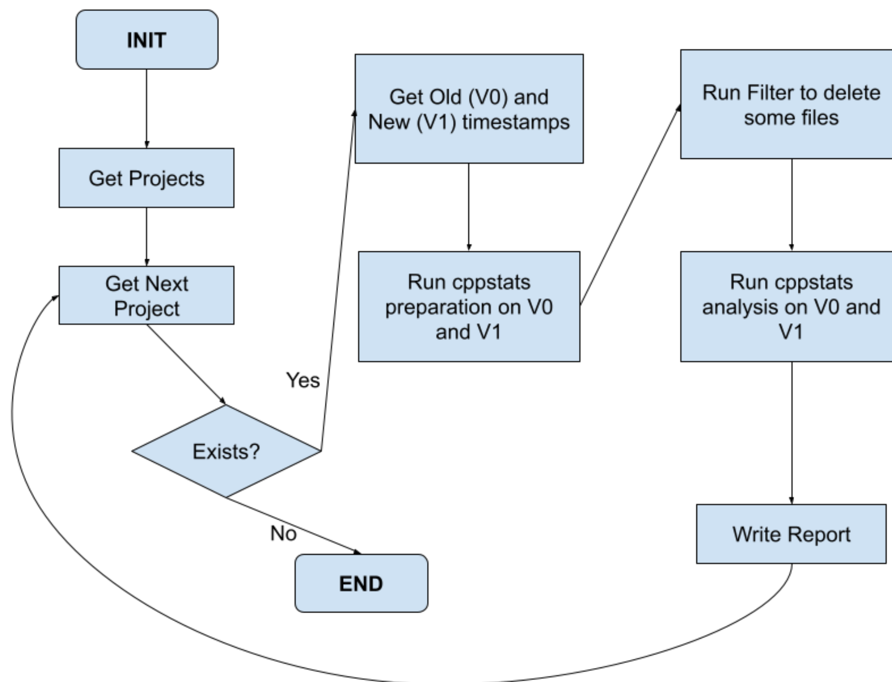


Figure 3.2: ifdef-catcher algorithm flow

research question on to what extent the number of disciplined annotations has increased or decreased over the last ten years. We then applied criteria to select projects that had a significant increase or decrease of disciplined annotations. These projects that had a significant increase of disciplined annotations, were project candidates to have refactorings. It helps us also to answer our second research question.

3.2.4 Analyze refactorings.

We developed a script based on the Malaquias‘ Heuristic[3] to collect candidates of refactorings Malaquias (2018). We used the project candidates to have refactorings that were defined in the previous step to execute the script. Figure 3.3 presents the flow of the script based on Malaquias’ heuristic. This heuristic identifies files that had an increase of one type of annotations and decreasing another type between a given commit (commit n) to a subsequent one (commit n+1). For each project, the script navigates through all commits, using cppstats to count and classify the annotations in each file. We then have the files that had at least one annotation that was disciplined between two commits, this annotation then can have been refactored between these commits. Thus, we then have a refactoring candidate with better accuracy. Once

[3]https://github.com/easy-software-ufal/ifdef-catcher/tree/main/phase-3-ifdef-transformation-detector/check_commits

the files are tagged, we performed a manual analysis to validate each refactoring candidate, answering then our second research question.

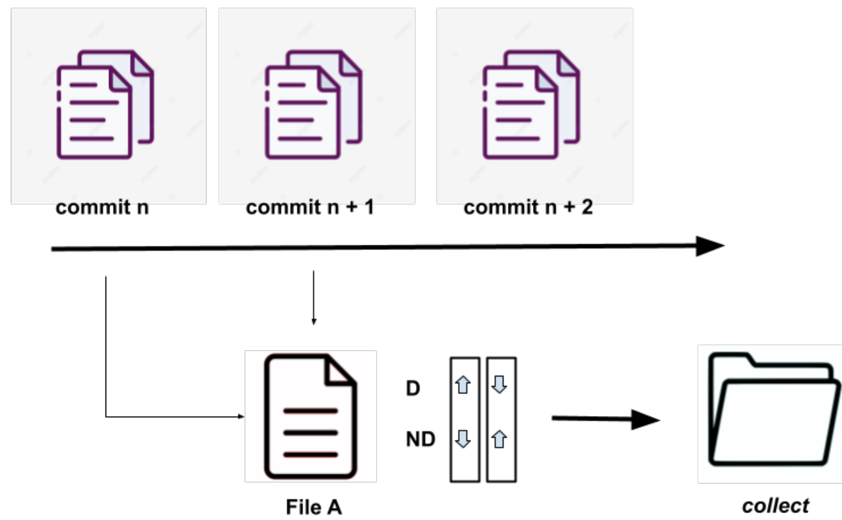
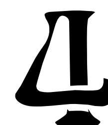


Figure 3.3: Malaquias Heuristic of Refactoring Candidates



Results

4.1 Frequency of Disciplined and Undisciplined Annotations along the last Ten years

This section presents the results of our first research question. The first question is about to what extent the number of disciplined and undisciplined annotations have increased or decreased over the last ten years.

Table 4.1 presents the results of the number of three metrics (lines of code, total of annotations and percentage of disciplined annotations) from 23 projects. Each metric has a value related to the old and a new version of each software project. For example, for project Apache, it is observed that, in column "Total of annotations", there was an 350 value in the first collected version, and a 387 value in the second collected version, where 37 is the difference of the two values.The column "Lines of code" denotes the code size of the project in terms of code lines. A little variation may mean that the project has changed little over the years. Regarding the column "Total of annotations", it indicates how relevant the pre-processor directives are for the project. If there is a positive variance, then it could mean that the project has come to rely more on annotations, and perhaps code disciplinarity will be more critical. If there was a negative variation, the project started to depend less on the annotations, and maybe disciplinarity will not be so important. The most important column in the table is "% disciplined". This column shows whether there has been an increase or decrease in disciplined code in the project. If the evolution is marked, then there is evidence that the project has seen significant changes in code discipline and perhaps more refactorings. For the apache project, there was a significant change in the lines of code (i.e. the project evolved a lot), the use of annotations grew, and code disciplinarity had a 4.3% change; this project is a good candidate for looking for refactorings. In the case of berkeley.db, for example, it also had a lot of change in the lines of code and the use of

annotations, but it seems to have not had much change in the code disciplinarity (0.22%), so it is not a good candidate.

	Lines of code			Total of annotations			% disciplined		
Projects	old	new	(new - old)	old	new	(new - old)	old	new	(new - old)
apache	35873	78287	42414	350	387	37	82.0%	86.3%	4.30%
berkeley.db	178003	252018	74015	3077	3823	746	92.46%	92.68%	0.22%
cherokee	4383	5926	1543	19	37	18	89.47%	100.0%	10.53%
clamav	17379	166726	149347	237	1439	1202	88.61%	90.48%	1.87%
cpython	155860	203773	47913	4643	4036	-607	96.58%	95.0%	-1.58%
dia	13937	17526	3589	46	80	34	95.65%	100.0%	4.35%
emacs	8654	42568	33914	322	2583	2261	94.41%	93.57%	-0.84%
freebsd	1632428	7601435	5969007	20155	70904	50749	91.67%	92.09%	0.42%
gcc	644729	2117635	1472906	4880	10729	5849	85.78%	88.39%	2.61%
gimp	119810	247393	127583	267	358	91	97.75%	96.09%	-1.66%
glibc	107781	230290	122509	2721	4093	1372	89.97%	93.43%	3.46%
gnumeric	59399	49814	-9585	792	94	-698	82.58%	90.43%	7.85%
gnuplot	3772	10190	6418	61	89	28	83.61%	88.76%	5.15%
libxml2	47647	58701	11054	1005	1070	65	98.81%	97.76%	-1.05%
lighttpd	3341	42278	38937	58	790	732	98.28%	92.03%	-6.25%
linux	2969690	16014484	13044794	20011	40877	20866	96.39%	97.17%	0.78%
openldap	9431	28539	19108	68	316	248	95.59%	92.09%	-3.50%
parrot	28164	45939	17775	582	278	-304	97.42%	98.56%	1.14%
php	192829	315432	122603	2640	4838	2198	90.08%	92.79%	2.71%
postgres	29494	250686	221192	199	1291	1092	92.46%	89.54%	-2.92%
sqlite	3424	127642	124218	43	1099	1056	93.02%	93.99%	0.97%
tcl	10192	24585	14393	262	607	345	95.42%	88.8%	-6.62%
xterm	1729	7780	6051	99	70	-29	94.95%	100.0%	5.05%

Table 4.1: Number of Disciplined and Undisciplined Annotations

To answer RQ1, we can look at the "% disciplined" column. For the difference between the old and new versions of each project, the mean of this column is 1.17, so it appears to be positive, that is, there seems to have been an increase in disciplined annotations over the 10 years. As the types of annotations are mutually exclusive and annotations can only be disciplined or undisciplined, this possible growth in disciplined annotations means a possible decrease in undisciplined annotations. However, we must perform a hypothesis test to see if this 1.17 mean is significantly greater than zero. A t-test was used in this case, since the sample is small and approximately normal. Figure 4.1 shows that the distribution is approximately normal, and a Shapiro-Wilk test considering an alpha value of 0.05 shows that there is sufficient evidence that the sample is normal, at least at the 95% confidence level. So, proceeding to the hypothesis test, we consider the following hypotheses:

$$\begin{cases} H_0\text{: the mean is less than or equal to zero.} \\ H_1\text{: the mean is greater than zero.} \end{cases} \tag{4.1}$$

The p-value of the test was 0.0929 (one tail), which is greater than the alpha value. That is, there is not enough evidence to reject the null hypothesis, so we cannot say, at the 95%

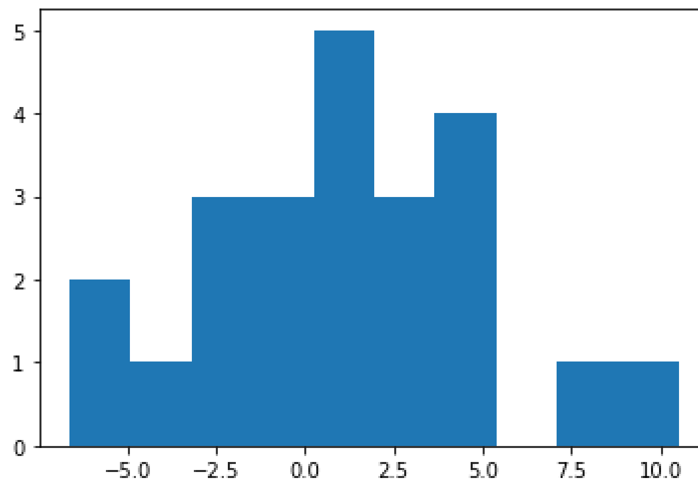


Figure 4.1: Distribution of changes in the percentage of disciplined annotations

confidence level, that the mean is significantly greater than zero. We cannot say that there has been a significant growth in the use of disciplined annotations in the last 10 years, just as we cannot say that there was a decrease in the use of undisciplined annotations. This may indicate that, at least, the use of disciplined and undisciplined annotations remained stable over the years, considering the analyzed projects.

Some projects had a variation well above the average in the "% disciplined" column (for example, the apache project). A question that can be asked is whether the developers of these projects were aware of the theory's existence, and the increase in disciplined annotations was due to this knowledge. Some projects also had a marked reduction in disciplined annotations; in these projects, what led you to move in this direction? Did the developers have some knowledge of the theory and still decided to take another path? These are questions that may be answered in future research.

For the next stage of the study, it was necessary to select one or more projects, from which the refactorings will be collected. The projects with the greatest variations in the column "% disciplined" (sub-column "new - old"), both positively and negatively, were selected. This is expected to find refactorings of both disciplined and undisciplined annotations. We used another criterion to have a better precision on projects that had refactorings. This criterion is based on a variation value of +3% on the increase of disciplined annotations, and -3% on the decrease of undisciplined annotation. The graph 4.2 presents the projects that had an increase of disciplined annotations (projects that had a variation marked by the green line) according to this criterion, and the projects that had a decreasing of disciplined annotations (projects that had a variation marked by the red line) according to this criterion. These projects were selected for the next phase of the study. According to this criterion, we then have ten projects that had a significant increase or decrease of disciplined annotations.

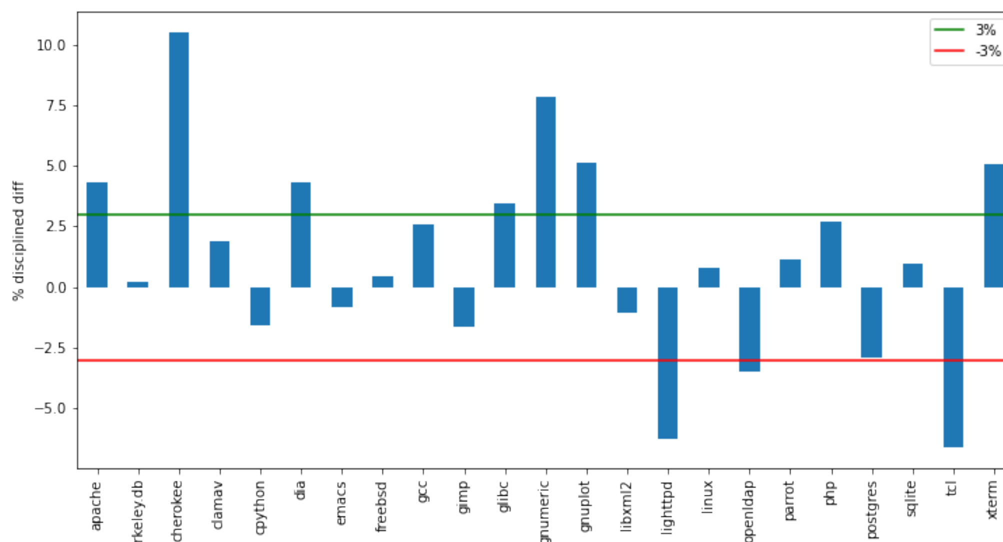


Figure 4.2: Projects with Refactorings Candidates

4.2 Refactorings on Undisciplined and Disciplined Annotations along the last Ten years

This section presents the results of our second research question. We then presented if developers have applied refactorings on undisciplined and disciplined annotations over the least ten years.

In the previous session, 10 candidates were chosen for refactorings to be extracted. After running the refactoring capture heuristic on these 10 projects, 19 refactorings were found. Table 4.2 summarizes the results by project. Note that some projects had no results; there are indications of what may have happened:

- cherokee: this project is small (5926 lines of code in the most recent version used) and has few annotations (37 in the most recent version used). Disciplinary variation of 10.53% may have been due to adding disciplined annotations or removing undisciplined annotations, or even both.
- tcl: This project has seen reasonable growth (14393 lines of code, with the addition of 345 annotations). What might have happened here is this: there was no effort to refactor annotations.

Most projects have had 1 or 2 refactorings. Most of the time, the refactorings found were well isolated and had no clues whether they were done to improve code readability. Even some refactorings were carried out to indiscipline a disciplined annotation, as can be seen in table 4.2.

All refactorings found can be seen on the github [1].

Projects	# of refactorings	# of ND -> D	# of D -> ND
apache	6	3	3
cherokee	0	0	0
dia	1	0	1
glibc	1	1	0
gnumeric	1	0	1
gnuplot	2	0	2
lighttpd	2	2	0
openldap	4	4	0
tcl	0	0	0
xterm	2	1	1

Table 4.2: Resume of refactorings found

```
#ifdef HAVE_THR
      switch ( fork1() )
#else
      switch ( fork() )
#endif
      {
      case -1:
        sleep( 5 );
        continue;

      case 0:
        break;

      default:
        _exit( EXIT_SUCCESS );
      }
```

Listing 4.1: Undisciplined annotation

```
#ifdef HAVE_THR
      pid = fork1();
#else
      pid = fork();
#endif
      switch ( pid )
      {
      case -1:
        sleep( 5 );
        continue;

      case 0:
        break;

      default:
        return pid;
      }
```

Listing 4.2: Disciplined annotation

[1] https://github.com/easy-software-ufal/ifdef-catcher/blob/main/phase-3-ifdef-transformation-detector/check_commits/refactorings_found.md

```
#ifdef GNM_USE_HILDON
  go_action_combo_text_set_width (
    wbcg->zoom,  "100000000%");
#else
  go_action_combo_text_set_width (
    wbcg->zoom,  "10000%");
#endif
```

Listing 4.3: Disciplined annotation

```
  go_action_combo_text_set_width (
    wbcg->zoom_haction,
#ifdef GNM_USE_HILDON
            "100000000%"
#else
            "10000%"
#endif
            );
```

Listing 4.4: Undisciplined annotation

To exemplify the two types of refactorings found (to discipline or indiscipline an annotation), we have chosen two cases to comment below.

In Listing 4.1, we have a **#ifdef** separating a **switch** from its body, which makes this annotation undisciplined. To discipline it, the variable **pid** was introduced inside **#ifdef**, to be used as a **switch** parameter (Listing 4.2). This way, the annotation does not break the **switch**. According to Medeiros et al. (2017), this refactoring is of the **wrapper** type.

In Listing 4.3, on the other hand, we have a disciplined **#ifdef** on the left side, where each statement is kept in one piece. The right-hand transformation (Listing 4.4) disrupts the directive, perhaps in order to avoid code repetition. This transformation makes the annotation undisciplined because it breaks the statement. In the Medeiros et al. (2017) catalog, this annotation is the inverse of the **comma-separated elements** type. The undisciplined annotation has its advantages (less code repetition, for example), although it can decrease code readability and therefore make maintenance difficult (Medeiros et al. (2015)).

More refactorings were found to discipline undisciplined code (11 refactorings) than indiscipline a disciplined code (8 refactorings). Furthermore, as can be seen in Figure 4.3, most results occur until 2016; this could mean that the importance of this type of refactoring has diminished over time, or that other solutions have been found to override annotations, or that code disciplinarity is no longer as relevant. These are all assumptions to be investigated further in future research.

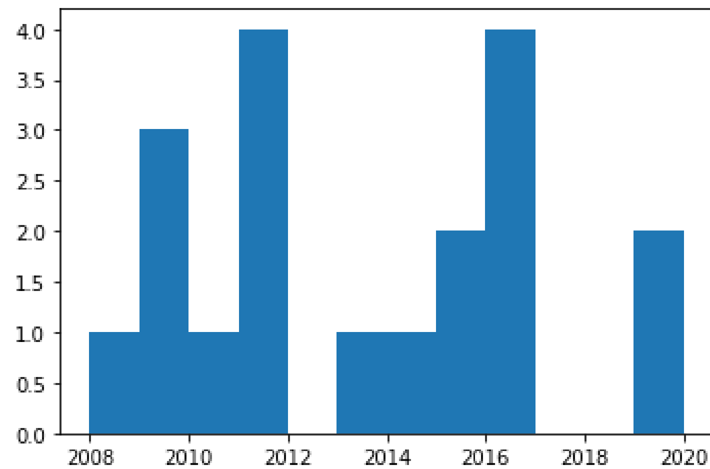


Figure 4.3: Refactors over time

Thus, the results found are indications that developers refactor not only undisciplined annotations, but also disciplined ones. What it is not possible to conclude, with just these results, is whether these refactorings were motivated by the disciplinarity theory or if they were motivated by other diverse issues. Future studies can answer these and other questions about refactorings linked to code disciplinarity. A more accurate selection of projects could have led to more refactorings. A suggestion for future research is to look at the characteristics of projects that had more refactorings, and establish more accurate criteria for selecting other candidates.

5

Threats to Validity

The projects were selected from Liebig et al. (2011), as they have a list of projects with metrics similar to what we were looking for in the analysis. However, we did not find some source codes or we did not find the latest version of some projects. We then avoided to replicate the study in every detail, using only a few methodologies to avoid having an inconsistent basis in relation to the original article.

In the selection of candidate projects for the search for refactorings, the variation in the use of disciplined annotations was considered as a criterion for this selection. However, only 19 refactorings were found in a set of 10 projects. Even not invalidating the analysis (some refactorings were found, and some conclusions could be drawn from these data), perhaps the selection criteria was not as accurate as we thought. In future research, it is recommended to refine this criterion to make it possible to select projects with more refactorings.

The heuristics to collect refactorings parse the entire file, and the tagged files are then handed over to manual analysis for refactorings. The problem is that some files are gigantic, which makes manual analysis very difficult. To minimize this problem, we used the github diff tool, which only shows the changes that have occurred in the file from one commit to another, making manual analysis much easier. As a suggestion for future research, perhaps handing in snippets of annotations for manual analysis will be more productive than handing in an entire file.

6

Conclusion

According to the results extracted from the analyzed projects, we observed that the amount of annotations remained stable over the years. Even after ten years or more, projects continue to use annotations at the same disciplined/undisciplined ratio. As it was necessary to select projects with greater evolution, the remaining option was to select projects with greater variations in the use of disciplined annotations.

The criteria used to select projects to collect refactorings led to 19 refactorings. For 10 projects, we believe that it may have been a result below expectations. Some projects had more refactorings found, and their characteristics may lead to better criteria for selecting projects in future research.

The refactorings found show that developers refactor both undisciplined and disciplined annotations. The results do not show, however, whether these refactorings are motivated by theory or by other reasons. So, this is a question that remains unanswered: did the developers have theoretical knowledge about annotation disciplinarity? Did the theory make an impact on the industry, or did it remain reclusive in academia? These are questions to be answered in future research.

References

Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 421–432, 2014.

Ira D Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 281–290. IEEE, 2001.

Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

Wolfram Fenske, Jacob Krüger, Maria Kanyshkova, and Sandro Schulze. # ifdef directives and program comprehension: The dilemma between correctness and preference. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 255–266. IEEE, 2020.

Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do# ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 65–73, 2016.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, 1999.

Duc Le, Eric Walkingshaw, and Martin Erwig. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150. IEEE, 2011.

Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 191–202, 2011.

Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. *ACM SIGOPS Operating Systems Review*, 40(4):191–204, 2006.

Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia, and Rohit Gheyi. The discipline of preprocessor-based annotations-does# ifdef tag n't# endif matter. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 297–307. IEEE, 2017.

Romero Bezerra de Souza Malaquias. A disciplinaridade das anotações condicionais de pré-processamento #ifdef tag não #endif importa. Master's thesis, Instituto de Computação - Universidade Federal de Alagoas, 2018.

Bill McCloskey and Eric Brewer. Astec: a new approach to refactoring c. *ACM SIGSOFT Software Engineering Notes*, 30(5):21–30, 2005.

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the c preprocessor: An interview study. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2017.

Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the discipline of preprocessor annotations matter? a controlled experiment. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 65–74, 2013.