# Analyzing the Evolution of Preprocessor-Based Variability: A Tale of a Thousand and One Scripts

Sandro Schulze
*Faculty of Computer Science*
*Otto-von-Guericke University Magdeburg*
Magdeburg, Germany
sanschul@iti.cs.uni-magdeburg.de

Wolfram Fenske
*Faculty of Computer Science*
*Otto-von-Guericke University Magdeburg*
Magdeburg, Germany
wfenske@iti.cs.uni-magdeburg.de

***Abstract*—Highly configurable software systems allow the efficient and reliable development of similar software variants based on a common code base. The C preprocessor CPP, which uses source code annotations that enable conditional compilation, is a simple yet powerful text-based tool for implementing such systems. However, since annotations interfere with the actual source code, the CPP has often been accused of being a source of errors and increased maintenance effort.**

**In our research, we have been curious about whether high-level patterns of CPP misuse (i.e., code smells) can be identified, how they evolve, and whether they really hinder maintenance. To support this research, we started a simple tool which over the years evolved into a powerful toolchain. This evolution was possible because our toolchain is not monolithic, but is composed of many small tools connected by scripts and communicating via files. Moreover, we reused existing tools whenever possible and developed our own solutions only as a last resort.**

**In this paper, we report our experiences of building this toolchain. In particular, we present design decisions we made and lessons learned, both positive and negative ones. We hope that this not only stimulates discussion and (in the best case) attracts more researchers in using our tools. Rather, we also want to encourage others to put emphasis on building tools instead of considering them *«yet another research prototype»*.**

***Index Terms*—source code analysis, software evolution, software maintenance, C preprocessor annotations**

## I. Introduction

Highly configurable software systems gained considerable attention in recent years as they allow to develop a whole family of related software systems of a specific domain. In this context, variability implementation mechanisms (e. g., feature-oriented programming [17], C preprocessor [1]) play a pivotal role, as they enable developers to introduce variability on code level, and thus, allow to maintain a single code base with integrated variability that encompasses *all* software variants.

In this paper, we focus on the C preprocessor CPP, as it is commonly used with C programs to introduce variability [11]. To this end, CPP directives such as `#ifdef` are used to annotate variable code that is removed in case the corresponding condition evaluates to false. Generally, the CPP is a language-independent tool constituting a powerful and expressive means to introduce source code variability. However, in the same way, the CPP has been considered harmful for lowering source code quality as it crosscuts first-class language elements, and thus, may impede program understanding [12, 19, 14, 13] or increase error-proneness (e. g., [21, 3, 2, 15]). While all of these shortcomings have been empirically validated, they are very specific, that is, they focus on a specific problem such as the discipline or misuse of particular CPP annotations, or specific syntactical errors. In contrast, we argue that they miss a more abstract reasoning about *how* and *when* CPP annotations should be considered harmful. As an example, we refer to the notion of code smells [7] that has been proposed and is widely accepted as higher-level patterns of bad coding practices.

In our research, we were curious whether such patterns of misuse also exist for CPP annotations. To this end, we developed a tool (along with the underlying concepts) to detect such patterns. Over the years, this tool evolved into a powerful tool chain that allows for flexible analysis of the impact of preprocessor annotations on the maintenance and evolution of highly configurable software systems. To arrive at this point, a couple of design and other decisions played a pivotal role, for instance, reliance on existing tools, techniques, and experiences, or avoiding a monolithic architecture.

In this paper, we report on our experiences in developing this tool chain. Besides simply presenting the tools and how they work in general, we rather provide insights into our (design) decisions and our reasoning when evolving the tool chain. Moreover, we present the lessons we learned during this journey. We hope to stimulate discussions about the importance of building proper tools instead of fragile one-time prototypes and about best practices for building tools.

## II. C Preprocessor in a nutshell

In this section, we briefly introduce how the CPP annotations are used for introducing variability and introduce the notion of variability-aware code smells.

### A. CPP *Variability*

The CPP has been shipped with the C programming language [9] from the very beginning as a text-based tool. Encompassing multiple directives (e. g., for macro definition), we are mostly interested in *conditional compilation* directives, which are commonly used for variability implementation in

This work was partly funded by project EXPLANT of the German Research Foundation (DFG, grant SA 465/49-1).

```
1  sig_handler process_alarm(int sig
2                            __attribute__((unused))) {
3    sigset_t old_mask;
4    if (thd_lib_detected == THD_LIB_LT &&
5        !pthread_equal(pthread_self(),alarm_thread)) {
6  #if defined(MAIN) && !defined(__bsdi__)
7      printf("thread_alarm in process_alarm\n");
8      fflush(stdout);
9  #endif
10 #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
11     my_sigset(thr_client_alarm, process_alarm);
12 #endif
13     return;
14   }
15 #ifndef USE_ALARM_THREAD
16   pthread_sigmask(SIG_SETMASK,&full_signal_set,
17                   &old_mask);
18   mysql_mutex_lock(&LOCK_alarm);
19 #endif
20   process_alarm_part2(sig);
21 #ifndef USE_ALARM_THREAD
22 #if !defined(USE_ONE_SIGNAL_HAND) && defined(
       SIGNAL_HANDLER_RESET_ON_DELIVERY)
23   my_sigset(THR_SERVER_ALARM,process_alarm);
24 #endif
25   mysql_mutex_unlock(&LOCK_alarm);
26   pthread_sigmask(SIG_SETMASK,&old_mask,NULL);
27 #endif
28   return;
29 }
```

Fig. 1. Example of a function with CPP annotatations, also suffering from the ANNOTATION BUNDLE smell [4]. Excerpt taken from MySQL, version 5.6.17, file `mysys/thr_alarm.c`.

practice [11]. Examples for such directives are, among others, `#ifdef`, `#ifndef`, `#elif`, or `#else` (cf. Figure 1). These directives are used to annotate variable code, that is, code that may be removed in a preprocessing step of the program (i. e., before compilation). To decide whether annotated code should be removed, a CPP directive is usually associated with a conditional expression (a. k. a. *feature expression*), which can either evaluate to true (code remains) or false (code is removed). Finally, a feature expression consists of either a single configuration option (a. k. a. *feature constant*) or constitutes a complex expression that is composed of multiple feature constants using logical operators (e. g., && or ||).

In Figure 1, we show a code snippet containing several kinds of CPP directives. For instance, Line 15 contains an expression with a single feature constant `USE_ALARM_THREAD`, whereas Line 22 contains a complex expression that consists of two constants, concatenated by a logical &&.

### B. Variability-Aware Code Smells

While being a popular tool for implementing variability, the CPP also received considerable criticism, because `#ifdef`s impede program understanding [3, 12, 19, 14, 13] and are prone to introduce subtle errors (e. g., [21, 2, 15, 14]). The reason is that CPP variability has several implications for the structure of source code (e. g. understandability, changeability), and thus, is a candidate for introducing code smells.

First, annotations are intrusive, which makes it hard to reenact the actual programming language statements in heavily annotated code. Second, composed feature expressions may encompass an arbitrary number of feature constants, which impedes reasoning about the conditions under which a certain piece of feature code is included. Third, code with CPP variability is prone to *scattering* and *tangling* as all feature code resides in a single code base. Both properties have been criticized for having a negative impact on understandability and changeability of annotated code [21, 2, 11].

All of these implications have been reported to occur repeatedly, and thus, heavily remind us of the original definition of code smells as *«recurring implementation patterns that are particularly harmful wrt. understandability and changeability»* [7]. However, so far only specific aspects of CPP variability had been taken into account for one or the other implication while more abstract patterns were missing. At this point, we introduced the notion of *variability-aware code smells* by adopting existing code smells and investigating how CPP annotations affect language elements [4]. In particular, we make use of the smell ANNOTATION BUNDLE for illustrative purposes in the remainder of this paper. This variability-aware code smell has been derived from the LONG METHOD smell and captures the extensive use of CPP directives in a method, thus, leading to many variable parts [4]. As an example, consider the function in Figure 1 that consists of many annotated code blocks, but with different CPP directives (with even different expressions), and thus, is likely to suffer from this smell.

## III. VARIABILITY-AWARE CODE SMELLS – THE SKUNK EXPERIENCE

After we proposed our variability-aware code smells (see the previous section for an overview), we wanted to validate their existence at a larger scale. While we had positive feedback from the academic community (by means of a survey; cf. [4]), a more quantitative analysis was necessary to validate our proposed smells. In particular, our aim was to demonstrate that our smells 1) occur frequently in practice and 2) cause problems, i. e., by hampering evolution or program comprehension.

However, after some research, we concluded that, no code smell detection tool so far had a focus on CPP annotations. This was the birth of SKUNK, our variability-aware code smell detector for large C programs.[1] Since we had only human-readable descriptions of our smells, we recognized that we had to answer two fundamental questions to guide our development:

- Q1 Which information should we employ to describe and, thus, detect variability-aware code smells?
- Q2 How can we extract instances of our smells efficiently, even from large systems?

In the remainder of this section, we elaborate on decisions we made to answer these questions and how they influenced the design of our tool.

### A. A Metrics-Based Approach

For Q1, we decided to employ *software metrics* to detect smells. This had mainly two reasons: 1) metrics have been commonly used for smell detection before [16, 22],

[1] https://github.com/wfenske/Skunk

and 2) most techniques that provide other information (e. g., control-flow) do not take CPP directives into account.

While a variety of metrics has been proposed in previous work (e. g., [12]), not all of them are applicable for each variability smell. For instance, for the ANNOTATION BUNDLE smell, we need metrics that address frequency as well as interference of CPP directives. As a result, we proposed the following metrics to be used together for detecting ANNOTATION BUNDLE: [5]

- *LOC* for source lines of code;
- *LOAC* for lines of *annotated code*;
- $ND_{acc}$ for *accumulated nesting depth*, that is, the sum of the nesting depths of all nested CPP directives (e. g., Line 21/22 in Figure 1);
- $NOFC_{dup}$ the number of feature constants in a function, including multiple occurrences;
- *NOFL* for *number of feature locations*, that is, the number of code blocks annotated with an `#ifdef` directive.

All of these metrics are combined into a formula (cf. [5]) and result in a value that indicates to what extent a function suffers from this smell. However, in different projects or for different developers, the perception which metric contributes more to a smell or which value is required to indicate a smell may differ. Hence, we allow for a *flexible parameterization* of these metrics by means of weights and thresholds [5]. To this end, our tool employs *code smell templates* (cf. Figure 2) in which the user can define thresholds for each metric to be considered. In particular, the decision for metrics and parameterization has been deeply inspired by previous work of Moha et al. and Van Emden et al. [16, 22].

**Lesson 1.** *Standing on the shoulders of giants.*
Learning from previous work by making use of foundational concepts and experiences is best practice and a key to overcome uncertainties and achieve acceptance.

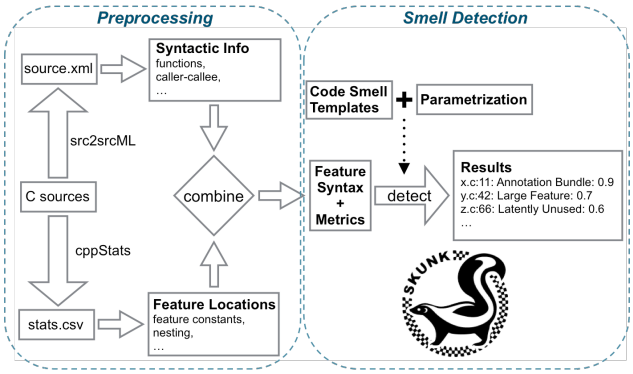


Fig. 2. Variability-aware code smell detection using SKUNK

Next, we present the overall architecture of SKUNK and how it collects information to compute the aforementioned metrics.

### B. The Tool Builder's Perspective

After we decided which information we should employ for the actual smell detection, we had to decide on how to collect this information. While for the ANNOTATION BUNDLE smell only lightweight metrics are needed, other smells encompass more profound information, such as control or even data flow, which requires more heavy techniques.

Fortunately, for previous analyses of CPP annotations, the tool CPPSTATS[2] has been proposed to collect basic information about occurrences and feature expressions of CPP directives. Moreover, this tool used SRCML[3] to obtain a bootstrapped AST of the code containing all CPP directives.

**Lesson 2.** *Prefer "proudly found elsewhere" over "not invented here".*
Many development efforts suffer from the "not invented here" syndrome: Instead of reusing mature third-party tools, an in-house solution is built, which then exhibits the same teething troubles that plagued many similar tools before it. Research prototypes are certainly no exception. Having some experience in professional software development, we tried to avoid this trap and instead embraced the "proudly found elsewhere" philosophy. Whenever possible, we built on existing tools that solved a specific subproblem we faced.

As a result of this important lesson learned, we integrated both mentioned tools into SKUNK, as shown in Figure 2. In particular, we use CPPSTATS to collect basic measures, such as annotated code blocks and corresponding expressions, and SRCML for collecting additional information, such as metadata about functions or caller-callee relationships.

Further design decisions, besides the ones already mentioned, were pivotal for the development and flexible use of SKUNK. First, we designed SKUNK with a kind of *Pipe&Filter* architecture. Initially, smell detection happened right after preprocessing, but we soon discovered that preprocessing was by far the most computationally step of our process. Hence, we decoupled both phases by persisting the intermediate data from preprocessing. The *Pipe&Filter* architecture made this an easy change. As a result, we were able to re-run our smell detection with different parameters without the need to execute the preprocessing again, which increased efficiency especially for large software systems. Besides facilitating changes, we argue that the *Pipe&Filter* architecture makes SKUNK easy to integrate in a larger tool chain.

Second, we designed SKUNK to be extended in the future. For instance, any further analysis for C source files can be combined with the current ones to establish information needed for detecting further code smells. Furthermore, we provide interfaces to make use of the results of the smell detection (i. e., the metric values). Hence, these results can be integrated into IDEs or used to visualize the findings.

[2]https://github.com/clhunsen/cppstats
[3]https://www.srcml.org/

**Lesson 3.** *Design for reuse and extension.*
Although an everlasting principle, its importance should not be underestimated. For us, reuse was crucial to create an efficient tool that can be easily tailored to project-specific properties. Moreover, design for extension ensures that new analyses can be added easily so that evolving the tool will not become a painful adventure.

## IV. Analyzing the Evolution of Preprocessor Annotations – the IfdefRevolver Experience

We used our SKUNK tool to investigate variability-aware code smells in five open-source systems [5]. This study provided insights into the frequency with which variability-aware code smells occur in practice and how they impact one important aspect of software development, namely program comprehension. But we were also interested in the impact of our smells on other aspects, such as maintainability and fault-proneness. Hence, we needed to analyze the evolution of a software system. Since SKUNK is unable to do that, we developed a second tool, IFDEFREVOLVER.[4]

### A. A Change-Based Approach

Our first goal with IFDEFREVOLVER was to investigate whether variability-aware code smells negatively affect change-proneness [6].[5] Again, we stood on the shoulders of giants when we devised the methodology to answer this question. Other researchers had already shown that change-prone code is also more fault-prone and requires more maintenance effort (e. g., [8, 20]). Moreover, we were aware of several studies that used this insight to investigate whether code smells have negative effects (e. g., [10, 18]). Following their methodology, we performed a study to investigate whether the presence of variability-aware code smells, specifically of ANNOTATION BUNDLES, negatively affects change-proneness. Since the ANNOTATION BUNDLE smell is specific to a C function, our question was basically, "Do functions that smell like ANNOTATION BUNDLES change more often or more extensively than other functions?"

The basic steps of IFDEFREVOLVER's analysis are:

1) Mine a source code repository to identify relevant commits and recreate a timeline from these commits
2) Slice this timeline into a series of *snapshots*
3) For each snapshot
   a) Identify all functions and locate the ones that are ANNOTATION BUNDLES
   b) Analyze how often/how much each function changes

After performing these steps, IFDEFREVOLVER provides the following information about a software system:

1) Which functions exist in each snapshot, and which of them are ANNOTATION BUNDLES?
2) How often does each function change within a snapshot?
3) How many lines are added to/deleted from each function?

We fed this information into various statistics (implemented in R) to determine whether the presence of the ANNOTATION BUNDLE smell affects the way that functions change. This seemed to be the case, but to our surprise, the difference was very small. More details on this study can be found in the corresponding paper [6]. In the next section, we discuss the lessons learned while implementing our tool.

### B. The Tool Builder's Perspective

The *Pipe&Filter* architecture that worked so well for SKUNK was also the blueprint for IFDEFREVOLVER. Nevertheless, the first prototypes were too monolithic, trying to solve too many tasks at once. This was a problem because changes became increasingly difficult and carried the risk of unwanted side effects. Hence, we embraced the Unix philosophy and repeatedly refactored IFDEFREVOLVER into ever smaller command line tools until each tool had a very simple, specific task. Each tool accepts a set of options, reads its input data from files, and also saves its results in files, which subsequently serve as input to other tools further down the pipeline. Redesigning IFDEFREVOLVER this way turned out to be very helpful, as we detail in the following lessons.

**Lesson 4.** *Do one thing, and do it well.*
In the spirit of classic Unix tools such as CUT or GREP, each of the tools that make up IFDEFREVOLVER only performs a small, well-defined task, but in combination, they accomplish great things.

We built tools that roughly correspond to the steps 1, 2, 3a, and 3b (cf. Section IV-A). Some tools are responsible for extracting information about the system under analysis (e. g., identifying relevant commits or extracting information about changed functions), whereas other tools are responsible for aggregating intermediate results into an overall result.

Splitting IFDEFREVOLVER into separate tools helped us maintain focus and clarity. It forced us to reason about what each tool was supposed to accomplish, which input data it needed and how results should be passed on along the pipeline. Once these questions were answered, implementing each tool became almost easy. Moreover, the risk of accidental breakage when fixing bugs or making enhancements was minimal, because most changes only affected a single tool.

Although performance-critical tools were implemented in JAVA, the overall analysis pipeline, which invokes the tools in the right order, is a BOURNE shell script. This script is supported by many additional scripts and some Makefiles, which, among other things, aggregate results, compute statistics, and generate figures. For this reason, the story of IFDEFREVOLVER is *a tale of a thousand and one scripts.*

As for SKUNK, we reused several third-party tools, which was easy due to IFDEFREVOLVER's modular structure. For example, detecting ANNOTATION BUNDLES during step 3a is as simple as checking out a revision of the analyzed system and running SKUNK on this checkout. The detection result is then fed to other tools for further processing. The important

[4] https://github.com/wfenske/IfdefRevolver
[5] https://wfenske.github.com/IfdefRevolver/ifdefs-vs-changes

point is that we were able to do that without changing SKUNK or making it aware of any evolutionary analyses.

**Lesson 5.** *Everything is a file.*
Instead of using databases, REST APIs or other forms of inter-system communication, many classic Unix tools operate on simple text files. We adopted this approach in IFDEFREVOLVER, making each tool read its input data from files and saving its results in files as well. Most output files constitute intermediate results, which serve as input for later stages of IFDEFREVOLVER's analysis pipeline. This approach created clean interfaces between our tools and made the dataflow easy to trace, which greatly helped development and debugging.

Every piece of information that our tools produces is saved in a file. For example, there is one file listing the commits that need to be analyzed. Then, there is another set of files detailing which commits belong to which snapshots. Likewise, for each snapshot, there is one file listing all functions that exist in this snapshot (specifying, e. g., the name of a function, lines of code, values of various annotation metrics), and another file that contains information about the number of times each function was changed and how many lines were added or removed. Aggregated results, too, are saved in files.

Most files are in CSV format because this format fits much of our data well and is easy to parse. Occasionally, we also use XML. During development, we frequently needed to check whether the right results were computed. Since we relied on textual formats, we could perform this check with a simple text editor – no need for debuggers or packet sniffers.

Alternatively, we could have employed a lightweight database, such as SQLITE. Besides enabling powerful SQL queries, this would also have lowered I/O cost. And in fact, we perform some computations with the help of SQL scripts. To this end, we use CSVKIT,[6] a set of command line tools that can treat CSV files as if they were database tables. I/O cost, by contrast, is not a performance bottleneck in IFDEFREVOLVER, because data extraction and preprocessing take up much more time. Thus, we benefit from database technology without having to deal with persistence APIs and other complexities.

The clarity of the interface and the dataflow that the file-based approach induced was very helpful when locating bugs: An erroneous result in a particular file could only be caused by the tool that created this file, or by errors in the tool's input files. The file-based approach also opened up opportunities to increase performance, which we detail in the next lesson.

**Lesson 6.** *Divide and conquer: Incremental and parallel analysis.*
Our analysis process consists of three main steps (see Section IV-A). Results from one step affect the results of subsequent steps, but never the other way around. Some steps can even run in parallel because they are independent of each other. Identifying these (in)dependencies had three implications: First, some computations could be parallelized, yielding a significant speed up on today's multi-core CPUs. Second, in case an error occurred, it was usually unnecessary to redo the whole analysis. Instead, only the erroneous part had to be repeated. Third, in case a file did not change in one snapshot, intermediate results for that file were reused in later snapshots.

By exploiting these insights, IFDEFREVOLVER's analysis became both incremental and parallel, which, compared to our initial implementation, reduced analysis times by up to an order of magnitude.

Especially during development, we frequently encountered situations that would break our tools and thus, interrupt an analysis. This was a problem because analysing a software system could take several hours. Fortunately, we found ways to avoid having to redo the whole analysis in most cases. One part of the solution was already in place, the fact that intermediate results are saved in files. The other part of the solution was to generate Makefiles, which specify how these result files depend on each other and which commands are necessary to create or update them. Thus, whenever a problem interrupted an analysis run, we would locate and fix that problem, remove invalid intermediate results, and simply start again. The MAKE tool would then recognize which result files were already up-to-date, which ones were outdated (because the inputs had changed), and which ones were still missing. Based on this information, MAKE would execute the necessary commands, effectively continuing the analysis just where it had left off.

In step 3a of IFDEFREVOLVER's analysis, CPPSTATS is run for each snapshot, which was initially very time-consuming. We observed that often only a few files change between one snapshot and the next, whereas the majority remains stable. Consequently, most of CPPSTATS computations were, in fact, redundant. Capitalizing on this observation, we enhanced CPPSTATS so that it reuses older results where possible and only computes new results when necessary.[7] This shift towards an incremental analysis greatly reduced analysis runtimes.

Modern versions of MAKE allow parallel builds, meaning that multiple files are compiled concurrently if they do not depend on each other. We use this feature in several places to parallelize our analysis. For example, each snapshot covers a different slice of the development timeline of the system under analysis. Consequently, most of the information extracted for a given snapshot is independent from other snapshots. Hence, multiple snapshots can be processed in parallel.

## V. CONCLUSION

C preprocessor annotations are frequently used to implement highly configurable software systems. Portions of the code base are annotated with conditional compilation directives, such as `#ifdef` or `#if`, and, thus, can be compiled into a specific variant or left out of it. Although widely used, several

[6] https://csvkit.readthedocs.io/

[7] https://github.com/wfenske/cppstats

studies have shown that CPP usage may cause problems. Complementing these studies, our research has focused on high-level patterns of annotation misuse, an idea that we call variability-aware code smells. Essential to this research were two tools that we built, SKUNK and IFDEFREVOLVER. Whereas SKUNK detects variability-aware code smells in one version of a software system, it can be combined with IFDEFREVOLVER to investigate how these smells evolve.

In this paper, we discussed the implementation of these tools, as well as the concepts behind them. More importantly, we shared six lessons learned while developing and enhancing these tools over several years. These lessons center around incorporating successful concepts from previous research, reusing existing tools, solving complex problems by solving many easy ones, and exploiting opportunities for parallel and incremental computation to speed up analyses. Although we learned these lessons while building tools to analyze CPP annotations, we believe they apply to many other domains.

## REFERENCES


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

[2] M. D. Ernst, G. J. Badros, and D. Notkin. "An Empirical Analysis of C Preprocessor Use". In: *IEEE Trans. Soft. Eng. (TSE)* 28.12 (2002), pp. 1146–1170.

[3] J.-M. Favre. "Understanding-in-the-Large". In: *Proc. Int'l Workshop on Program Comprehension (IWPC)*. IEEE, 1997, pp. 29–38.

[4] W. Fenske and S. Schulze. "Code Smells Revisited: A Variability Perspective". In: *Proc. Int'l Workshop on Variability Modeling of Software-intensive Systems (VaMoS)*. ACM, 2015, pp. 3–10.

[5] W. Fenske, S. Schulze, D. Meyer, and G. Saake. "When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells". In: *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 171–180.

[6] W. Fenske, S. Schulze, and G. Saake. "How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness". In: *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2017, pp. 77–90.

[7] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering". In: *IEEE Trans. Soft. Eng. (TSE)* 38.6 (2012), pp. 1276–1304.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., 1978.

[10] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness". In: *Emp. Soft. Eng.* 17.3 (2012), pp. 243–275.

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

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

[13] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. "The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter". In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2017, pp. 297–307.

[14] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. "The Love/Hate Relationship with the C Preprocessor: An Interview Study". In: *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 495–518.

[15] F. Medeiros, M. Ribeiro, and R. Gheyi. "Investigating Preprocessor-Based Syntax Errors". In: *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2013, pp. 75–84.

[16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. "DECOR: A method for the specification and detection of code and design smells". In: *IEEE Trans. Soft. Eng. (TSE)* 36.1 (2010), pp. 20–36.

[17] C. Prehofer. "Feature-Oriented Programming: A Fresh Look at Objects". In: *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Springer, 1997, pp. 419–443.

[18] D. Romano and M. Pinzger. "Using Source Code Metrics to Predict Change-Prone Java Interfaces". In: *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 2007, pp. 303–312.

[19] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. "Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment". In: *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM. 2013, pp. 65–74.

[20] D. I. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba. "Quantifying the Effect of Code Smells on Maintenance Effort". In: *IEEE Trans. Soft. Eng. (TSE)* 39.8 (2013), pp. 1144–1156.

[21] H. Spencer and G. Collyer. "#ifdef Considered Harmful, or Portability Experience With C News". In: *Proc. USENIX Technical Conference*. USENIX Association, 1992, pp. 185–197.

[22] E. Van Emden and L. Moonen. "Java quality assurance by detecting code smells". In: *Proc. Working Conf. Reverse Engineering (WCRE)*. IEEE, 2002, pp. 97–106.