When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells

Wolfram Fenske
University of Magdeburg, Germany
wfenske@ovgu.de

Sandro Schulze
TU Braunschweig, Germany
sanschul@tu-bs.de

Daniel Meyer, Gunter Saake
University of Magdeburg, Germany
Daniel3.Meyer@st.ovgu.de, saake@ovgu.de

***Abstract*—Code smells are established, widely used characterizations of shortcomings in design and implementation of software systems. As such, they have been subject to intensive research regarding their detection and impact on understandability and changeability of source code. However, current methods do not support *highly configurable software systems*, that is, systems that can be customized to fit a wide range of requirements or platforms. Such systems commonly owe their configurability to conditional compilation based on C preprocessor annotations (a. k. a. `#ifdef`s). Since annotations directly interact with the host language (e. g., C), they may have adverse effects on understandability and changeability of source code, referred to as *variability-aware* code smells. In this paper, we propose a metric-based method that integrates source code and C preprocessor annotations to detect such smells. We evaluate our method for one specific smell on five open-source systems of medium size, thus, demonstrating its general applicability. Moreover, we manually reviewed 100 instances of the smell and provide a qualitative analysis of its potential impact as well as common causes for the occurrence.**

I. Introduction

Code Smells, as introduced by Fowler et al., describe shortcomings in source code caused by improper design or evolution of a software system [10]. In particular, a code smell represents a concrete pattern in the use of language mechanisms and elements that indicates deeper problems in the underlying source code or design. Specifically, code smells may have a negative effect on program comprehension [1], maintenance [32], and evolution [16] of software systems. Hence, it is desirable to detect *and* correct such smells to increase the overall quality of the respective system. Based on Fowler's human-readable patterns, a vast amount of methods have been proposed to automatically detect (and partially to correct) such smells. Among others, visual exploration [25], metric-based detection, pattern-based detection [31, 17], and rule-based detection [24] have been proposed and evaluated.

However, all of these approaches fall short when dealing with *highly configurable software systems*, that is, systems that exhibit *variation points* and thereby allow the development of a whole family of related programs, tailored to different sets of requirements [6]. These variation points (a. k. a. *features*) distinguish program variants by their differences while, at the same time, allowing to reuse parts that are common among programs, thus, improving time-to-market and reliability compared to single system development. Different variability mechanisms are used to implement variation points, for instance feature-oriented programming [26] or conditional compilation by means of C preprocessor annotations [3].

In this paper, we focus on the C preprocessor, CPP, as it is widely used in C programs to introduce variability [19]. Basically, CPP directives, such as `#ifdef`, are used to annotate configurable code fragments, which are removed (and consequently not compiled) if the corresponding preprocessor condition evaluates to false. While this enables developers to flexibly express variability even at a fine-grained level, it comes at the cost of directly interfering with the actual source code (e. g., the C program). Consequently, when used in an inappropriate way, CPP annotations may impede program understanding, hinder changeability or even foster the introduction of syntactical and semantical errors [22].

While commonly considered both, harmful yet necessary, the impact of CPP has been only analyzed with respect to certain aspects such as error proneness [7] or disciplined usage [21], while a more general, pattern-based analysis is still missing. In previous work, we presented a methodology to derive variability-aware code smells from traditional, single-system smells, as well as an initial catalog of such smells for feature-oriented programming and preprocessor variability [8]. However, an empirical evaluation about existence and nature of such code smells has not been provided so far.

In this paper, we now propose a metric-based method to detect variability-aware code smells and evaluate it on mature and widely used open-source software systems. In particular, we make the following contributions:

- A set of metrics that captures shortcomings in variability implementation and thus, enables the detection of variability-aware code smells in C programs.
- A tool, called SKUNK, implementing our metric-based method to detect the aforementioned code smells. Among other capabilities, our tool allows for parametrization of thresholds and weighting factors for individual metrics.
- An empirical study for the variability-aware code smell ANNOTATION BUNDLE. To this end, we analyzed five software systems with up to 285 KLOC. Besides quantitative evaluation, we manually reviewed 100 instances of this code smell to a) assess the usefulness of our detection method and b) provide insights into how such smells manifest themselves in source code.

While our study indicates points for improvement of our detection method, it also shows that variability-aware code smells

1) commonly exist in C programs and 2) tightly interact with the host language. Hence, we argue that such smells indicate problems, impeding changeability and understandability of the source code in highly configurable software systems.

In the remainder, we provide foundations of the CPPas variability mechanism in Sec. II. Then, we present our concept of variability-aware code smells and their automated, metric-based detection. Sec. IV, we present design and results of our case study, while Sec. V provides details of our qualitative analysis. Finally, we end with related work (Sec. VI) and a conclusion (Sec. VII).

II. Boon and Bane of Preprocessor Variability

In this section, we describe how preprocessors are used to implement variability in software and how this affects the structure of source code. First, we describe how variability can be implemented using preprocessors, in particular, using the C preprocessor, CPP. Then, we point out the effects of preprocessor variability on the structure of source code, and, finally, we discuss how these effects are related to code smells.

A. Implementing Variability Using the C Preprocessor

The CPP tool, originally invented for the C programing language [15], is a general-purpose, text-based preprocessor. The way CPP preprocesses text is controlled by directives, for instance for macro definition (`#define`) or file inclusion (`#include`). More important to our discussion is *conditional compilation*, which is widely used in both, academia and industry, for implementing variability [3]. Examples for such directives are `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`. In a nutshell, the implementation of a feature (a. k. a. *feature code*) can be annotated with these directives, and, during preprocessing, annotated code is conditionally removed for a specific program variant. Whether to remove feature code or not depends on whether the conditional expression of an `#if` (or `#elif`) is true or false. We call this expression a *feature expression*. The most common form of feature expressions is to test whether a name, which we call a *feature constant*, is defined as a preprocessor macro or not. In fact, this form is so common that directives of the form `#if defined(X)` can be abbreviated to `#ifdef X`. Hence, we will synonymously use `#ifdef` to refer to any conditional compilation directives in the rest of the paper. Simple feature expressions can be combined to form more complex ones using logical operators (such as `&&` or `||`). Although not considered in our case, feature expressions can not only be boolean but also arithmetic, for instance, by comparing the value of a macro to a numeric constant.

We illustrate the use of CPP conditional compilation by means of a "Hello world" example shown in Fig. 1. Depending on how this code is compiled, three distinct program variants can be created. The first variant simply prints "Hello world!"; the second additionally asks the user to guess a number and always congratulates her for guessing right; the third variant also asks for a number, but always claims the user guessed wrong. The inclusion of the corresponding feature code (Lines 4, 8–9, 11, and 13) depends on the feature constants `GUESS_POS` and `GUESS_NEG`. Specifically, if `GUESS_POS` or `GUESS_NEG` are defined, Lines 4 and 8–14 are further processed by CPP. In contrast, if neither feature constant is defined, these lines are excluded. It is possible to nest `#ifdef`s, in which case the processing of the nested `#ifdef` depends on the enclosing one. In Fig. 1, the `#ifdef` on Line 10 is nested and thus, is only evaluated if the surrounding `#ifdef` (Line 7) evaluates to true.

```
1  #include <stdio.h>
2  int main(int argc, char **argv) {
3  #if defined(GUESS_POS) || defined(GUESS_NEG)
4    int x;
5  #endif
6    printf("Hello world!\n");
7  #if defined(GUESS_POS) || defined(GUESS_NEG)
8    printf("What is my favorite number? ");
9    scanf("%d", &x);
10 #ifdef GUESS_POS
11   printf("Yes, %d is my favorite number!\n", x);
12 #else
13   printf("No, %d is my favorite number!\n", x+1);
14 #endif
15 #endif
16   return 0;
17 }
```

Figure 1: A "Hello world" program, comprising three different program variants due to preprocessor variability.

B. Boon and Bane

Code with `#ifdef`s is often criticized as being hard to understand and susceptible to the introduction of subtle, hard-to-spot bugs [30, 7, 23, 22]. Despite the frequent criticism, CPP is a popular tool to implement variability. Amongst other, the reasons include that the annotate-and-remove model is easy to understand, requires little pre-planning, and obviates the need for extra tool support as CPP is part of the compiler [13].

Preprocessor variability has several implications for the structure of source code that are related to understandability and changeability and, thus, to code smells. First, annotations are intrusive. As each preprocessor directive requires an entire line of text, it can become hard to spot the actual programming language statements in heavily annotated code. As an example, the program in Fig. 1 contains only five lines of feature code but seven lines of annotations. Second, composed feature expressions can refer to an arbitrary number of feature constants and `#ifdef`s can be nested. Both cases can lead to difficulties when reasoning about the conditions under which a certain piece of feature code is included. In our example, to determine whether Line 11 is compiled, we not only have to consider the `#ifdef` on Line 10, but we also need to look at the `#ifdef` on Line 7. Third, code with preprocessor variability is prone to *scattering* and *tangling* as all feature code resides in a single code base. Scattering means that feature code occurs in multiple locations, whereas tangling means that feature code of one feature is mixed with other feature code or with base code. Both, scattering and tangling, have been criticized as having a negative impact on understandability and changeability of

#ifdef code [30, 7]. For instance, the code of feature GUESS_POS is scattered to Lines 4, 8–9, and 11. Hence, in order to understand how GUESS_POS is implemented, all of these locations have to be considered. Moreover, the code of GUESS_POS and GUESS_NEG is tangled with each other and with the base code. For this reason, the implementation of GUESS_POS can neither be understood nor changed without also considering GUESS_NEG. Finally, preprocessors allow for fine-grained extensions, such as adding statements in the middle of a function or annotating individual expressions within a statement [14]. This can be seen as an advantage over other variability mechanisms, such as plug-in architectures or aspect-oriented programming, which require complicated workarounds to allow extensions with this level of granularity. However, fine-grained extensions have also been criticized as a source of subtle bugs [4, 30].

C. Variability-Aware Code Smells

The aforementioned effects, such as scattering, tangling, or composed feature expressions, affect the structure of source code, thus, impeding understandability and changeability of source code. For first-class programming language entities (e. g. functions or classes), Fowler et al. have introduced the concept of code smells to describe recurring implementation patterns that are particularly harmful wrt. understandability and changeability [10]. For instance, the LONG METHOD smell has been proposed to characterize methods that implement too much functionality, without abstracting solutions to sub-problems into smaller pieces (i. e., methods). As a result, such a method is hard to understand and maintain. Considering variability implementation, improper use of CPP annotations may have similar effects as we have shown in previous work [8]. We refer to such shortcomings as *variability-aware code smells,* that is, code smells that specifically occur in the presence of variability.

III. DETECTING VARIABILITY-AWARE CODE SMELLS

As presented in the previous section, variability has an impact on how source code is structured and can therefore lead to variability-aware code smells. In this section, we propose a metric-based method for detecting variability-aware code smells in C programs using preprocessor variability. Specifically, we describe the characteristics of the ANNOTATION BUNDLE smell, the metrics we propose to detect this smell, as well as our implementation of a variability-aware code smell detection tool, called SKUNK. Note that, while our tool already computes metrics for other smells, proposed in [8] as well, we focus on ANNOTATION BUNDLE, because a) these metrics have been validated and b) it allows for a more in-depth, qualitative analysis of the detection results.

A. Annotation Bundle

As proposed in previous work [8], ANNOTATION BUNDLE is the variability-aware version of the established LONG METHOD smell. While LONG METHOD is a function that contains too much functionality, ANNOTATION BUNDLE is a function that contains too much variability. In particular, an ANNOTATION BUNDLE manifests itself as a function in which many (groups of) statements are annotated with #ifdef directives. Moreover, feature expressions of those #ifdefs may refer to several feature constants, can involve negation, and may be nested.

```
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 }
```

Figure 2: Example of a function suffering from the ANNOTATION BUNDLE smell. Excerpt taken from MySQL, version 5.6.17, file mysys/thr_alarm.c.

In Fig. 2, we show an example of a function indicating such a smell in the open source database management system MySQL, version 5.6.17.[1] With 29 lines of code in total, this function is not overly long. However, what makes this function an ANNOTATION BUNDLE is the amount and appearance of preprocessor variability. Given 29 lines in total, only 19 lines are actual C code, while the remaining ten lines are preprocessor directives. Moreover, of these 19 lines of C code, nine lines are feature code, leaving only ten lines of base code. Hence, only half of the function's implementation is static, while the other half is only optionally available in a concrete program. This feature code is grouped into five feature locations (e. g., Lines 6–9 and 10–12). Also, feature expressions refer to five unique feature constants (e. g., MAIN and __bsdi__), two of which occur twice. Negation is used three times (Lines 6, 15, and 22), and there is one nested directive, starting on Line 22. Finally, in Lines 6 and 22, complex feature expressions are used, whereas the other feature expressions are atomic (i. e., refer to one feature constant).

We argue that a function, such as the one in Fig. 2, is difficult to understand as the code is obfuscated by the large

[1] http://dev.mysql.com/downloads/file.php?id=451519

Table I: Atomic metrics, capturing basic characteristics of the ANNOTATION BUNDLE code smell

Abbreviation	Full Name	Description
LOC	Lines of code	Source lines of code of the function, ignoring blank lines and comments.
$LOAC$	Lines of annotated code	Source lines of code in all feature locations within the function. Lines that occur in a nested feature location are counted only once. Again, blank lines and comments are ignored.
ND_{acc}	Accumulated nesting depth	Nesting depth of annotations, accumulated over all feature locations within the scope. An `#ifdef` that is not enclosed by another `#ifdef` is called a *top-level* `#ifdef` and has a nesting depth of zero; an `#ifdef` within a top-level `#ifdef` has a nesting depth of one, and so on. Nesting values are accumulated, which means that a function containing two feature locations with a nesting depth of one is assigned an ND_{acc} value of 2.
$NOFC_{dup}$	Number of feature constants, including duplicates	Number of feature constants, accumulated over all feature locations within the scope. Feature constants that occur in multiple feature locations are counted multiple times.
$NOFL$	Number of feature locations	Number of blocks annotated with an `#ifdef`. An `#ifdef` containing a complex expression (e. g. `#ifdef A && B`) counts as a single feature location. An `#ifdef` with an `#else`/`#elif` branch counts as two locations.

amount of features and how they are encoded [8]. Moreover, change dependencies may arise if several feature locations are controlled by the same feature constant, which impairs changeability. For instance, the feature locations on Lines 10 and 22 both refer to `SIGNAL_HANDLER_RESET_ON_DELIVERY`. Due to this relation, changes to the feature expression or the comprised source code in one of these locations imply co-changes to the other location.

B. Metrics

In previous work, we proposed human-readable descriptions of variability-aware smells, which limits automatic detection [8]. In this section, we outline our method to automatically detect variability-aware code smells in C programs using CPP directives for variability. Although our method is designed to detect a variety of smells, we solely focus on the detection of ANNOTATION BUNDLEs in this paper.

Proposed metrics: Our detection method is based on metrics that are statically extracted from source code. The atomic metrics that we use to detect ANNOTATION BUNDLEs are given in Tab. I. Apart from the LOC metric, simply counting lines of code, these metrics are specifically designed to capture the complexity added by preprocessor annotations. For instance, $LOAC$ counts the lines of feature code within a function, whereas $NOFL$ measures the number of feature locations.

As our ANNOTATION BUNDLE is affected by several of the aforementioned metrics, we propose an aggregated metric, AB_{smell}, to obtain a value that indicates to what extent a particular function suffers from this smell. We show the resulting formula in Eq. 1.

$$w_1 \cdot \frac{LOAC}{LOC} \cdot NOFL + w_2 \cdot \frac{NOFC_{dup}}{NOFL} + w_3 \cdot \frac{ND_{acc}}{NOFL} \quad (1)$$

This formula consists of three terms, intended to capture three aspects of complexity introduced by `#ifdef` annotations: Term one computes the ratio of annotated code (feature code) to all code. The reasoning behind this ratio is our assumption that, for instance, 10 lines of feature code in a function with a total length of 20 lines have more of a negative effect than 10 lines of feature code in a function of 100 lines. Furthermore, based on the intuition that a function with several small feature locations is more problematic than a function with a large, single feature location, the $LOAC$ to LOC ratio is multiplied by the number of feature locations. As a result, with increasing scattering of feature code, the value of term one also increases. The second term is designed to capture the average complexity of feature expressions within a function. If all feature expressions consist only of a single feature constant, term two evaluates to one. However, if complex feature expressions are used, (i. e., two or more feature constants), term two will evaluate to a value greater than one. Finally, the third term accounts for nesting. Without nesting, this term evaluates to zero. In contrast, having two feature locations, and the second location is nested within the first one, the value will be 0.5.

For our example in Fig. 2, the values of the atomic metrics are as follows: $LOC = 29$, $LOAC = 9$ (nine lines of annotated code), $NOFL = 5$ (five feature locations), $NOFC_{dup} = 7$ (three feature constants are unique, while two occur twice), and $ND_{acc} = 1$ (nesting only occurs once, for the feature location on Line 22). Next, we use these metrics to compute the terms of AB_{smell}, which yields 1.55, 1.4, and 0.2 for the first, second, and third term, respectively. Assuming weights w_1, w_2, and w_3 to be 1, the final AB_{smell} value for the function in Fig. 2 is approximately 3.15.

Parameterization and thresholds: How much a particular atomic metric contributes to the complexity, perceived by a human developer, may vary depending on individual preferences. Hence, our detection method allows to parametrize these metrics using *weights* and *thresholds*. For the AB_{smell} metric (see Eq. 1), the relative influence of each term is adjustable using the weights w_1, w_2, and w_3. Moreover, thresholds can be specified as *lower* boundary for a particular metric, thus, reducing false positives in the result set. For instance, a user could use a threshold of $LOAC\ ratio = 0.5$ in order to only consider functions with at least 50 % of annotated code. As a result, if intimate knowledge of the source code or guidelines for CPP usage are available, users can encode this knowledge in the provided weights and thresholds to obtain more precise results.

C. Implementation

In Fig. 3, we show the basic detection process of SKUNK, a tool implementing our variability-aware code smell detection method. Detection starts with a preprocessing step, regarding the source code of the analyzed system. Pre-

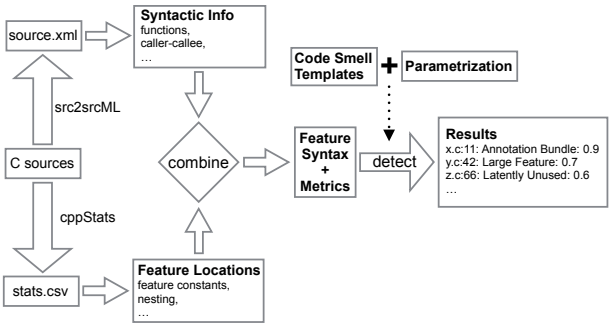

Figure 3: Variability-aware code smell detection using SKUNK

processing is performed by two external tools, CPPSTATS[2] and SRC2SRCML[3] [5]. CPPSTATS extracts most variability-related information that SKUNK requires, such as the locations of `#ifdef` directives. However, SKUNK needs additional, syntatic information (e. g., the location of function definitions), which CPPSTATS does not provide. To extract this information, we rely on SRC2SRCML, which transforms C code into an XML representation containing information similar to an AST.

After preprocessing, SKUNK extracts feature locations and other variability-related information from the results of CPPSTATS, while function definitions and further metrics are extracted from the output of SRC2SRCML. Next, SKUNK combines both sources of information in order to calculate the atomic metrics shown in Tab. I. The results of this calculation are stored in an internal format for further processing (cf. **Feature Syntax + Metrics** in Fig. 3).

Based on these metrics, the actual detection of variability-aware code smells takes place. Controlled by a parameterizable *Code Smell Template*, SKUNK searches for functions that match the user-specified parameters. For instance, the user may want to limit reported ANNOTATION BUNDLEs to functions that contain at least 50 % of annotated code. This can be achieved by setting parameter `Method_LoacToLocRatio` to `0.5;mandatory`, where `0.5` is the threshold value and keyword `mandatory` instructs SKUNK to ignore functions with a ratio below that threshold. For functions that pass this filtering step, SKUNK computes the AB_{smell} metric (see Eq. 1). Afterwards, the metric value, along with the function's name and location, is written to a result file (cf. **Results** in Fig. 3). Finally, the results can be inspected by the user, thus, identifying functions of interest and take appropriate corrective action.

Moreover, additional parameters, not considered here, exist, which can be set to user-specified values in order to achieve more precise results. Beyond that, our tool computes further metrics to be used in the future to detect additional smells. The source code of SKUNK, as well as a short HowTo and examples, can be obtained from our complementary webpage.[4]

[2]http://www.fosd.net/cppstats/

[3]http://srcml.org/

[4]https://www.isf.cs.tu-bs.de/cms/team/schulze/material/scam2015skunk/

Table II: Overview of subject systems used in evaluation

Name	Version	Domain	LOC	% LOAC	AB_{pot}	AB_{smell}
Emacs	24.5	Text editor	247,403	29.0	20	6.9
Libxml2	2.9.2	XML processing lib	215,751	69.5	76	4.0
Lynx	2.8.8	Text-based browser	115,102	43.8	76	10.9
PHP	5.6.9	Scripting language	117,813	18.2	26	5.1
Vim	7.4	Text editor	285,817	69.8	259	7.1

Version, Domain: version number and application domain. **LOC:** number of non-blank, non-comment lines of C code, ignoring header files. **% LOAC:** ratio of lines of annotated code (feature code) to total lines of code, in percent. **AB_{pot}:** number of potential ANNOTATION BUNDLES reported by SKUNK. **AB_{smell}:** median of metric values, computed over all candidates of AB_{pot}.

IV. CASE STUDY

Given the introduced notion of ANNOTATION BUNDLE, it is of superior interest to investigate to what extent this smell occurs "in the wild". To this end, we apply our proposed detection in an exploratory case study using five highly configurable open-source software systems. The main goal of this study was to establish a ground truth for the detection of ANNOTATION BUNDLES but also to develop an infrastructure for detecting further variability-aware smells (cf. [8]). In the remainder of this section, we present the setup of our case study, including research questions, subject systems, methodology, and results.

A. Research Questions

Basically, we want to answer two research questions aimed at assessing the usefulness of both, the concept of variability-aware code smells as well as our detection method.

RQ 1: Does our algorithm detect meaningful instances of the ANNOTATION BUNDLE *smell?* With this question, we evaluate the precision of our detection algorithm. By means of manual inspection, we assess to what extent automatically detected instances of ANNOTATION BUNDLE align with human perception regarding understandability and changeability.

RQ 2: Does the ANNOTATION BUNDLE *smell exhibit recurring, higher-level patterns of usage?* With this question, we aim at investigating the reason developers introduced the smell. In particular, we want to know whether implementation- or domain-specific characteristics foster the occurrence of the smell. Moreover, we investigate which of such patterns are more likely to negatively impact the underlying source code.

B. Subject Systems

In Tab. II, we show an overview of the subject systems for our exploratory case study. We used five highly configurable open-source systems of medium size. For each system, we list its version, domain (e. g., text editor), size (in LOC; measured using the CLOC tool[5]), the LOAC ratio (in %), and the number of potential instances of the ANNOTATION BUNDLE detected by SKUNK. Note that values in LOC and LOAC column only include non-blank and non-comment lines of C code. Moreover, both measures do not include header files, since SKUNK currently does not analyze them. We report the LOAC

[5]http://cloc.sourceforge.net

ratio (column **% LOAC**) to provide a rough impression of how much variable code each system contains.

Our subject systems constitute a subset of forty systems used by Liebig et al. to analyze preprocessor usage [19]. We based our selection on three criteria: 1) We chose systems of medium size in order to get a sufficient number of potential instances of the ANNOTATION BUNDLE smell but also to show the applicability to systems of reasonable size. 2) We targeted systems with a considerable fraction of annotated code, the rationale being that systems with a high proportion of feature code are more likely to contain ANNOTATION BUNDLES than systems with little feature code. 3) In order to remove bias regarding specific properties of a particular application domain, we chose systems of four different domains. Nevertheless, we included two text editors (*Emacs* and *Vim*) to account for different coding styles within a single domain.

Threats to validity: Our selection of subject systems may be not representative for other domains, such as embedded systems. However, as systems represent different domains, our selection is not biased to only one specific domain, and our results can at least be generalized to similar domains.

C. Methodology

In the following, we describe our methodology for detecting and evaluating the ANNOTATION BUNDLE smell, which consists of three steps: parametrization of the detection algorithm, creating the sample set, and validating these samples.

Parametrizing the detection algorithm: As mentioned in Section III, the user can parametrize the detection process by providing thresholds and weighting factors for particular measures. For our case study, we set thresholds for the following four parameters:

- $LOAC\ ratio = 0.5$: As the considered smell focuses on problematic usage of CPP directives, we required each function to contain at least 50% annotated code. Otherwise, we assumed to obtain too many false positives, that is, functions that only partially exhibit smelliness.
- $NOFC = 2$: Since tangling and scattering are part of our AB_{smell} metric, we required the occurrence of at least two different feature constants.
- $NOFL = 2$: We argue that one large chunk of annotated code is something that happens frequently in C programs and hardly indicates problems for changeability or understandability. Thus, we chose a higher threshold.
- $ND = 1$: Our intuition was that nesting significantly contributes to the complexity of an ANNOTATION BUNDLE. Hence, we set this threshold such that only functions with at least one nested `#ifdef` directive are reported.

All of the aforementioned parameters are mandatory for the detection process, that is, a function will only be reported by SKUNK if it adheres to *all* of these thresholds. Based on test runs of SKUNK on other system, we chose these thresholds to reduce the number of false positives. While we could have chosen higher thresholds, this might have resulted in decreasing recall, that is, the likelihood of missing potentially smelly functions would have increased.

Sample selection: After running SKUNK on our subject systems, the question arises to what extent reported smell instances are false positives or not. Since there is no baseline for variability-aware code smells so far (e. g., a human oracle), we decided to manually inspect a subset of the detected smells (as inspecting all smells is infeasible in a reasonable time). We took a sample of 20 smells from each subject system by the following method: Half the samples (i. e., 10 per system) constitute those smells with the highest AB_{smell} value. As a result, we can also evaluate to what extent the metric value is a good indicator for a code smell. For the second part of the sample set, we split the remaining result set into 10 equally distributed segments. For each of those segments, we randomly selected one entry and added it to our sample set. This way, we obtained a cross-selection of all smells for our evaluation. Moreover, we gain insights on a possible borderline regarding our detection metric, that is, whether smells occur only above a certain metric value. For example, the detection result of *Lynx* consists of 76 potential smells. In step one, we included the top-10 entries in our sample set.Next, we split the remaining 66 entries into 10 segments, encompassing approximately seven entries each, and randomly selected on entry from each segment This sample set was then subject to manual validation, which we explain next.

Validating the smells: We use the previously created sample set for assessing the precision of our detection algorithm. To this end, two authors of this paper performed *cross validation* by reviewing each sample independently and assigning a rating to it, thus, simulating a human oracle (not available otherwise). We use a 3-point scale for assessing the sample smells with the ratings -1 (no negative impact), 0 (partial negative impact), and 1 (high negative impact). While the first indicates a false positive sample (i. e., no smell), the latter are considered to be at least partial smells. The difference between partial and high impact of smells is that the former indicates that only certain code fragments of the sample impede changeability and understandability while for the latter, the whole (or most of) the function is negatively affected.

For the actual inspection of the smells (and their impact), we mainly focused on how much preprocessor annotations interfere with data and control flow. Amongst others, this includes annotated variable initialization/assignments, annotated conditional branches (both, whole blocks as well as just parts of the condition) and loops. Moreover, we consider nested annotations and compound feature expressions (i. e., `#ifdef` directives with multiple feature constants). For all of these occurrences, the authors evaluated how difficult annotations make it to understand the concrete functionality (e. g., follow data / control flow) or identify the location for a possible change. Note that we mainly disregarded shortcomings that result *only* from inappropriate use of the programming language, such as sequences of nested conditionals or just overly long functions. Based on this review process, each of the reviewing authors assigned a rating to a respective smell sample. Moreover, they recorded qualitative data, such as why or why not a certain sample constitutes a smell, what is

Table III: Detection results for ANNOTATION BUNDLE smell

Name	AB_{-1}		AB_0		AB_{+1}		AB_{0+1}		% Prec.	
	top10	all	top10	all	top10	all	top10	all	top10	all
Emacs	2	8	4	8	4	4	8	12	80	60
Libxml2	3	12	3	4	4	4	7	8	70	40
Lynx	2	8	2	6	6	6	8	12	80	60
PHP	6	15	4	5	0	0	4	5	40	25
Vim	1	6	1	4	8	10	9	14	90	70
Total	14	49	14	27	22	24	36	51	72	51

AB_{-1}: samples with a manual rating of -1 (no impact). **AB_0:** samples with a manual rating of 0 (medium impact). **AB_{+1}:** samples with a manual rating of $+1$ (high impact). **AB_{0+1}:** samples with a manual rating of 0 or $+1$ (medium or high impact). **% Prec.:** ratio of manually confirmed smell instances rated 0 or $+1$ in relation to sample size, in percent. The *top10* subcolumns refer to the top ten detection results. The *all* subcolumns refer to all twenty samples.

the actual functionality provided, and which are reasons for introducing variability.

Finally, both reviewing authors compared their results, smell by smell. In case of different ratings, they discussed their ratings (including the qualitative observations) until they achieved a rating both authors could agree on.

Threats to validity: While we designed the above methodology with care, it inherently comprises certain threats. First, parametrization may lead to missing smells in the results set. However, we have chosen conservative values for the respective parameters, thus mitigating the risk of missing "real" smells. Second, sampling bears the risk not being representative or too small regarding the whole data set. We mitigate this threat in two ways: 1) Our selection method incorporates peak values (i. e., highest value for AB_{smell} metric) as well as a coverage of the value domain (using a randomized, equally distributed sampling strategy). 2) Our sample sets encompass between 8% and 100% of the initial result set, thus having a reasonable size for being representative. Finally, our validation process may introduce a bias, because the ratings are based on personal opinions of the reviewing authors, which renders the results subjective. Hence, we set up a strategy with concrete criteria for rating the inspected smells *in advance*, thus, mitigating the effect of rating potential smells at random. Moreover, the authors rated the smells independently in first place and discussed their results in a final review meeting.

D. Results

The results of the manual inspection and validation are given in Tab. III, while we provide the initial result set, containing all potential instances of ANNOTATION BUNDLE after running our detection algorithm, in Tab. II.

Initial result set: Basically, our data reveal that each of the subject system exhibits potential smells. Nevertheless, the amount of detected smells differs considerably, ranging from only 20 (for *Emacs*) to 259 (for *Vim*). These differences indicate that there is considerable heterogeneity in *how* different systems use CPP annotations. Moreover, the AB_{smell} metric value (computed as median over all values per system; cf. Tab. II) exhibits differences as well. Hence, for each system, a different cut-off value may apply at which the result set turns from rather reliable results into rather false positives.

Inspection of sample set: In Tab. III, we show the results for our manual inspection of the sample set. In particular, we show two different views on our results: ratings for the whole sample set (stated in the *all* subcolumns) and for a reduced sample set encompassing only the smells with the ten highest AB_{smell} metric values (stated in the *top10* subcolumns). For the whole sample set, our data reveals that approx. 50 % of the detected smells have been discarded (rating of -1). While this indicates only an average precision of our detection algorithm, the data also reveals that each system exhibits smells that have a medium or high impact on changeability and understandability of the source code (according to our manual validation). Moreover, precision differs considerably between systems, ranging from 25 % (for *PHP*) to 70 % (for *Vim*). For the top-10 sample set, in contrast, we obtain more precise results. In particular, the average precision is 72 %, with four systems having a precision between 70 % and 90 %. Additionally, note that the number of smells with high impact (rating $+1$) clearly exceeds the rating for medium impact, thus, providing a higher confidence that the detected smells may constitute a problem in source code. Hence, we argue that especially the smells with high metric values are good indicators for shortcomings and thus, are useful to guide developers to code locations (i. e., functions) that require special attention with respect to program understanding or code changes.

We provide detailed results (e. g., code artefacts, metric values, and ratings) on our complementary web page. Next, we present results and insights of our qualitative analysis.

V. QUALITATIVE ANALYSIS

In this section, we report more on qualitative observations that we made during our manual inspection of the samples. In particular, we relate these observations to our research questions (cf. Sec. IV-A) and provide details why smells occur, whether they follow certain patterns, and why there are sometimes good reasons for how developers use annotations.

A. RQ 1: When is a Smell a Smell?

The goal of RQ 1 is to find out whether our algorithm detects meaningful instances of the ANNOTATION BUNDLE smell. Overall, we made four important observations with respect to this question, which we provide in the following. *Basically, these observations show that our metric-based approach enables the detection of smells that have a negative impact.* However, we also gained a better understanding on possible weaknesses of our algorithm. For instance, our algorithm disregards recurring patterns of annotations that actually improve code understandability, thus, not constituting smells.

Observation 1: *There is no single reason for why functions smell.* During manual inspection of potential smells, we repeatedly noted that no single property (e. g., LOAC ratio, nesting depth) reliably predict whether a sample was actually negatively affected by variability (according to the author's perception). Rather, combining different parameters

is necessary to make a potential smell a real one. For instance, samples 10 and 14–20 of *Libxml2* constitute unit tests for configurable (thus optional) functionality. Since these tests would not compile if the corresponding functionality is absent, most of the body of these tests is enclosed in few but large annotated blocks. Hence, these test functions exhibit a high percentage of annotated code (approximately 95 %), indicating a highly variable function. However, there are only two feature locations, and annotations are very coarse-grained. Consequently, these samples were rated -1 by both authors, as they where easy to understand.

We learned from this observation that our metric, which combines several parameters, is a good baseline for detecting ANNOTATION BUNDLES in many real-world scenarios. Nevertheless, as the *Libxml2* samples show, there is room for improvement since particular patterns of `#ifdef` usage can lead to false positives.

Observation 2: *Interactions of preprocessor variability with runtime variability leads to smelly code.* Another observation we made is that annotated code amplifies the negative impact of already complex control flow. For instance, function `ins_redraw` from *Vim* (sample 13, 83 LOC) is rather short. Nevertheless, it contains 11 (runtime) `if` statements, and most `if`s are nested over one or two levels within another `if`. While this control flow is already hard to understand, these `if` statements are additionally annotated with 12 (compile-time) `#ifdef`s. Even more, some of these `#ifdef`s are used to annotate parts of the `if`-condition. As such complex interactions impede understandability and changeability, this example and similar ones have been rated $+1$ (high impact).

We learned from this observation that structural properties of *how* CPP directives interfere with the host language are an important ingredient for a more precise detection algorithm.

Observation 3: *Keep it short.* Many short functions (i. e., $LOC \leq 100$), even some with a top-10 AB_{smell} value, were have been rated -1 (no impact). Longer functions, in contrast, were more likely to be rated 0 or $+1$ (medium/high impact). The reasons may be related to the interaction between complex code in the host language and preprocessor variability, as discussed in Observation 2. Short functions are less likely to contain complex control flow. Hence, they inherently exhibit fewer interactions between preprocessor and runtime variability. An exemplary comparison of the LOC metrics for *Lynx* confirms our observation; samples rated -1, 0, and $+1$ have an average length of 83, 294, and 319 LOC, respectively. We observed a similar correlation (though different numbers) for *Emacs*. Both examples indicate that LOC may be positively correlated with preprocessor annotations that have a negative impact. Hence, we learned that the precision of our AB_{smell} metric may benefit from taking LOC into account, although counterexamples show that LOC alone is not sufficient.

Observation 4: *Repetitive feature code aids comprehension.* Some smells have been discarded although they exhibited high AB_{smell} values (e. g., due to a vast amount of feature locations and constants). The reason is that we observed a repetitive structure of the variable code, which aided comprehension

```
1  static char *(has_list[]) =
2  {
3  /* ...*/
4  #ifdef OS2
5    "os2",
6  #endif
7  #ifdef __QNX__
8    "qnx",
9  #endif
10 #ifdef UNIX
11   "unix",
12 #endif
13 /* ... more features ... */
14 };
```

Figure 4: Example for repetitive feature code in *Vim*

when reading the code. For instance, function `f_has` in *Vim* (file `src/eval.c`) contains 95 % of annotated code and 177 feature locations. This function received the highest AB_{smell} value for *Vim*. However, most feature locations in `f_has` are related to a single statement that initializes an array used to query the configuration of a particular *Vim* installation at runtime. As we show in Fig. 4, feature locations are basically an enumeration of all configuration options provided by *Vim*. Consequently, function `f_has` has been rated -1, because the repetitive structure aids understanding and changing (i. e., adding new features) the code. Note that our observation is in line with the work of Kapser and Godfrey regarding the benefits of code cloning practices that are not necessarily considered harmful [12]. In this sense, we argue that repetitive feature code is another argument in favor of their claim.

In summary, we learned that recurring structural similarities of annotated code should be considered for our detection algorithm in order to increase precision.

B. RQ 2: High-Level Patterns of Annotated Code

With RQ 2, we investigate whether more abstract patterns exist that induce the ANNOTATION BUNDLE smell, or even prevent annotated code from being a smell. Based on our qualitative analysis, we identified three of such patterns and report another observation that may introduce further smells.

Observation 5: *Adapter pattern.* Some platforms (e. g. operating systems or compilers) lack common functions or provide implementations that cause problems. Hence, systems that rely on these functions provide adapters for platforms that do not provide a suitable implementation. For instance, *Emacs* provides an adapter for `gettimeofday`, whose purpose is to return the current time. On some platforms, this adapter will delegate to the platform-provided implementation and perform clean-up actions to work around a bug. On other platforms, this adapter will delegate to another function, `ftime`, which provides similar functionality but a different interface. In the particular case of *Emacs'* `gettimeofday` adapter, the code was rated 0, indicating only partial negative impact on some parts of the function. In general, we learned that adapters are beneficial with respect to overall code quality as they protect surrounding source code from the variability they encapsulate.

Observation 6: *Optional Feature Stub.* We found a number of functions doing nothing if a particular optional feature is

not present. We call this pattern OPTIONAL FEATURE STUB. The unit tests for *Libxml2*, described in Observation 1, are one example. As a further example, *Emacs* contains a function to initialize settings of the *fontconfig* library. In case *fontconfig* support is disabled, this function can still be called, but it will have no effect. Although OPTIONAL FEATURE STUBs have a LOAC ratio of almost 100%, most samples that followed this pattern were rated -1. In particular, we argue that OPTIONAL FEATURE STUBs are beneficial, because callers of the stub remain oblivious of a particular feature being present or not. Hence, we learned that the OPTIONAL FEATURE STUB actually reduces variability-related complexity and consider them as beneficial patterns rather than being smells.

Observation 7: *Featurized God Function.* Especially in *Vim*, we found many functions with more than 1000 LOC (e. g., samples 2, 3, 4). Beyond their sheer length, these functions contained a lot of variable code, usually annotated by a vast amount of different feature constants. For instance, function `win_line` (*Vim*, sample 2, 2153 LOC) contains 23 feature locations related to multibyte strings, 22 feature locations for right-to-left writing systems, and 16 feature locations related to syntax highlighting, among others. In the style of the anti-pattern GOD CLASS [27], we call these functions FEATURIZED GOD FUNCTIONS, because they encompass a huge degree of (diverse) variability. Although samples from other systems constitute long functions as well, we particularly identified *Vim* as being prone to this style of functions. As a result, we learned that too much variability related to too many features is a common pattern for the ANNOTATION BUNDLE smell as it impedes understanding of data and control flow.

Observation 8: *Other platform and library variability.* Generally, we found platform- and library-related differences being a frequent source of variability. For instance, differences in representing path names on Windows and Unix-like operating systems were a reason for variable code in *Lynx* (samples 8, 9, and 11), *PHP* (samples 4 and 8), and *Vim* (sample 15). Moreover, differences between alternative libraries or different versions of the same library foster the introduction of variability. For instance, *Libxml2* can either use the *iconv* or the *uconv* library to convert text between different character encodings. Although the interfaces of *iconv* and *uconv* are similar, they are not identical. Hence, the *Libxml2* function `xmlFindCharEncodingHandler` (sample 11) contains alternative feature code for each library.

VI. RELATED WORK

A arge body of research addresses metrics and tools to detect smells and anti-patterns in object-oriented software [31, 18, 17, 24]. Furthermore, Figueiredo et al. presented a detection approach for modularity flaws based on metrics and concern-sensitive heuristics [9]. Moreover, Abilio et al. discuss metrics for smell detection in feature-oriented programming [2]. Additionally, we have investigated the relationship of DUPLICATED CODE and preprocessor variability [28]. All this work is related to ours as it addresses the automatic detection of code anomalies. In contrast, this work either investigates smells and anti-patterns in single software systems [31, 18, 17, 24], focusses on cross-cutting concerns [9], on only one smell [28], or on detecting smells in configurable systems using academic variability mechanisms [2]. In contrast, we present a general detection method for variability-aware code smells, focus on a commonly used variability mechanism, and provide an evaluation on real-world systems.

Many researchers investigated the use of CPP and its potential negative impact on source code. Specifically, Spencer and Collyer argued against the use of `#ifdef`s for building portable software, based on experience with a single system [30]. Ernst et al. empirically analyzed CPP usage patterns, but focus on potential problems of macro expansion and techniques to replace CPP usage [7]. However, neither Spencer and Collyer, nor Ernst et al. address automatic detection of code anomalies. Next, Medeiros et al. investigated bugs related to preprocessor variability [23, 22]. In particular, he found that developers perceive variability-related bugs easier to introduce, harder to fix, and more critical than other bugs [22]. This finding underlines the importance of our work, which investigates sources of complexity introduced by preprocessor variability. Other researchers have analyzed scattering, tangling, and nesting (as well as other properties) of `#ifdef` directives in highly configurable systems [19, 11]. In contrast to our work, their analyses are statistical in nature, and do not discuss methods to detect concrete patterns of misuse.

Code smells serve as a guideline for when and where to refactor source code [10]. We present an approach to automatically identify code fragments in highly configurable systems that will likely benefit from refactoring. Hence, our work is complementary to our previous work on variability-aware refactoring [29], as well as recent advances in automating refactoring in the presence of CPP directives [20].

VII. CONCLUSION

Variability plays a pivotal role for developing highly configurable software systems in a fast and reliable way, because it supports structured reuse of existing and tested code. Conditional compilation using C preprocessor annotations is a common mechanism to implement variability in source code. Due to the increasing importance of highly configurable software, the effect of preprocessor annotations on source code quality and perception of developers is gaining attention as well. In particular, improper use of preprocessor annotations may introduce *variability-aware code smells*, that is, patterns that negatively affect understandability and changeability of source code. In this paper, we proposed a metric-based technique to detect such smells and applied it to five highly configurable software systems from different domains. We validated our technique by performing a comprehensive manual inspection of the automatically detected smells. Furthermore, we qualitatively investigated reasons for the occurrence of smells. Overall, we obtained two main results: First, we observed code smells in each subject system, which shows that they are a common phenomenon in variable software systemsSecond, our qualitative analysis revealed that certain structural properties

affect the impact of such smells, but also that certain patterns exist that render apparently smelly code beneficial.

In future work, we intend to improve our detection algorithm based on the lessons learned in the presented study. Moreover, we aim at an extended validation of the impact of such smells by either conducting experiments or interviews with developers of the considered systems. Finally, we will investigate whether such smells correlate with other peculiarities, such as being changed more frequently or being more prone to the introduction of bugs.

References

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension". In: *CSMR*. IEEE, 2011, pp. 181–190.

[2] R. Abilio, J. Padilha, E. Figueiredo, and H. Costa. "Detecting Code Smells in Software Product Lines – An Exploratory Study". In: *ITNG*. 2015, pp. 433–438.

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

[4] I. D. Baxter and M. Mehlich. "Preprocessor Conditional Removal by Simple Partial Evaluation". In: *WCRE*. IEEE, 2001, pp. 281–290.

[5] M. L. Collard, H. H. Kagdi, and J. I. Maletic. "An XML-Based Lightweight C++ Fact Extractor". In: *IWPC*. IEEE, 2003, pp. 134–143.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

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

[8] W. Fenske and S. Schulze. "Code Smells Revisited: A Variability Perspective". In: *VAMOS*. ACM, 2015, pp. 3–10.

[9] E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena. "Applying and Evaluating Concern-Sensitive Design Heuristics". In: *J. Soft. Sys.* 85.2 (2012), pp. 227–243.

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

[11] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. "Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study". In: *Emp. Softw. Eng.* (2015), pp. 1–34.

[12] C. Kapser and M. W. Godfrey. ""Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software". In: *Emp. Softw. Eng.* 13.6 (2008), pp. 645–692.

[13] C. Kästner and S. Apel. "Virtual Separation of Concerns – A Second Chance for Preprocessors". In: *J. Object Technology* 8.6 (2009), pp. 59–78.

[14] C. Kästner, S. Apel, and M. Kuhlemann. "Granularity in Software Product Lines". In: *ICSE*. ACM, 2008, pp. 311–320.

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

[16] F. Khomh, M. Di Penta, and Y. Gueheneuc. "An exploratory study of the impact of code smells on software change-proneness". In: *WCRE*. IEEE, 2009, pp. 75–84.

[17] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. "A bayesian approach for the detection of code and design smells". In: *QSIC*. IEEE, 2009, pp. 305–314.

[18] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[19] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines". In: *ICSE*. ACM, 2010, pp. 105–114.

[20] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. "Morpheus: Variability-Aware Refactoring in the Wild". In: *ICSE*. ACM, 2015, pp. 380–391.

[21] J. Liebig, C. Kästner, and S. Apel. "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code". In: *AOSD*. ACM, 2011, pp. 191–202.

[22] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. "The Love/Hate Relationship with the C Preprocessor: An Interview Study". In: *ECOOP*. to appear. 2015.

[23] F. Medeiros, M. Ribeiro, and R. Gheyi. "Investigating Preprocessor-Based Syntax Errors". In: *GPCE*. ACM, 2013, pp. 75–84.

[24] 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. Softw. Eng.* 36.1 (2010), pp. 20–36.

[25] E. Murphy-Hill and A. P. Black. "An Interactive Ambient Visualization for Code Smells". In: *SOFTVIS*. ACM, 2010, pp. 5–14.

[26] C. Prehofer. "Feature-Oriented Programming: A Fresh Look at Objects". In: *ECOOP*. Springer, 1997, pp. 419–443.

[27] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[28] S. Schulze, E. Jürgens, and J. Feigenspan. "Analyzing the Effect of Preprocessor Annotations on Code Clones". In: *SCAM*. IEEE, 2011, pp. 115–124.

[29] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. "Variant-Preserving Refactoring in Feature-Oriented Software Product Lines". In: *VAMOS*. ACM, 2012, pp. 73–81.

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

[31] E. Van Emden and L. Moonen. "Java quality assurance by detecting code smells". In: *WCRE*. IEEE, 2002, pp. 97–106.

[32] A. Yamashita. "How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study". In: *ICSM*. IEEE, 2013, pp. 566–571.