University of Magdeburg

School of Computer Science

Master's Thesis

Metrics-Based Code Smell Detection in Highly Configurable Software Systems

Author:

Daniel Meyer

November 30, 2015

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Department of Technical and Business Information Systems

Dipl.-Inf. Wolfram Fenske

Department of Technical and Business Information Systems

Dr.-Ing. Sandro Schulze

Institute of Software Engineering and Automotive Informatics
Technische Universität Braunschweig

Meyer, Daniel:
Metrics-Based Code Smell Detection in Highly Configurable Software Systems
Master's Thesis, University of Magdeburg, 2015.

Abstract

A *code smell* is a symptom in computer programming that may indicate design flaws or code decay within a software system. As such, much research has been conducted regarding their detection and impact on understandability and changeability of source code. Current methods, however, can not be applied to *highly configurable software systems*, that is, variable systems that can be configured to fit a wide range of requirements or platforms. This variability is often based on conditional compilation implemented using `C` preprocessor annotations (`#ifdef`). These annotations directly interact with the host language (e.g., `C`) and therefore may have negative effects on understandability and changeability of the source code. In this case, we refer to them as *variability-aware code smells*. In this thesis, we propose 1) a concept of how to detect such variability-aware code smells with a metrics-based approach, and 2) a code smell detector, called Skunk, that employs this concept. In our evaluation, we use the detector on 7 open-source system of medium size to find potential variability-aware code smells samples, on which we perform a manual inspection on 20 samples of each subject system. Our results show an average precision of 40.7% (Annotation Bundle) and 62.1% (Annotation File) for detecting actual code smells. For the Large Feature we used a statistical approach to decide at which point features implement an excessive amount of functionality. The results show that features with more than 1122 lines of code are an uncommonly large. Additionally, we proposed severity ratings for ranking potential code smells, which proved to be a good indicator for the smelliness of samples. Higher-rated samples are more likely to have a negative effect on understandability and changeability than low-rated ones. For each code smell, we examined frequently recurring patterns across all systems.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

A *code smell* is a symptom in computer programming that may indicate design flaws or code decay within a software system [Fow99]. Each code smell defines a specific pattern in the use of language mechanisms and elements that hints at deeper problems in the underlying source code or design. In particular, code smells may have a negative effect on program comprehension [AKGA11], maintenance [Yam13], and evolution [KPG09] of software systems. Code smells can be detected by using metric-based detection [VEM02][KVGS09], pattern-based detection [MGDLM10], visual exploration [MHB10], and rule-based detection. A developer employs such methods to detect and eliminate harmful smells manually or by refactoring [KPG09][Fow99]

A newer programming concept in industry and research is the development of *highly-configurable software systems* (or *software product-lines*) [ABKS13]. It allows developers to create multiple products out of a product family in a fast and efficient manner. Each instance is based on a shared amount of features, where each product is defined by a different selection of features. A feature describes a specific function or concern within the product family. One possibility to implement features in software systems are annotation-based methods with the help of preprocessors [ABKS13]. The code that belongs to a feature will only be compiled for the system if the feature is selected. This method is used in numerous open source and industrial systems such as PHP, Vim or MPlayer [LAL+10].

While classic object-oriented code smells (for instance, *Long Methods*) exist in these software systems, other types of code smells can be identified within the source code as well. Fenske and Schulze [FS15] described new variability-aware code smells which are a result of the variability inside a highly-configurable software system. Moreover, they have shown that experienced software developers observed variability-aware code smells (or variability smells) while working with software product-lines. However, due to the novelty of this research field, there is no method for detecting code smells automatically. Consequently, we lack empirical evidence about how frequently they occur in practice.

Therefore, I propose a concept and a tool to identify variability-aware code smells in highly configurable software using annotation-based variability. Furthermore, I perform a case study to prove its applicability and gather empirical data on how frequently variability-aware code smells occur in practice.

To this end, I implement a new code-smell detector specialized on C code with CPP annotations, called Skunk. Code smell detection via Skunk consists of two steps, *metrics extraction* and *metrics analysis*. In the first step, the tool processes the source code to get important metrics (e.g., *lines of code*) of all features of the system. This metrics data set can be saved to the disk and is used to quicken up the detection process later. During the *metrics analysis* step, the metrics data set will be analyzed to detect variability-aware code smells in the source system. The characteristics of a code smell can be described by using the metrics extracted during the first phase. The tool uses customizable configuration files, in which the user is able to set thresholds and ratios to define the minimal requirements for a code smell.

The generated detection output is separated into multiple files. The tool generates severity reports for files, methods and features in CSV files. These files present the basic metrics for the object. Furthermore, it contains code smell values which are aggregated from the basic metrics. These code smell values serve as indicators for the severity of code smells. In addition, Skunk generates location files, which contain exact position information for the code smells. With the help of these generated files, I am able to find code smells, decide if they are in fact code smells and thus evaluate the precision of Skunk.

With the help of Skunk, I conduct a case study. In this study, I process different software systems to get the metrics of all features. Then, I perform the detection with configuration files that define the code smells Large Feature and Annotation Bundle/File [FS15]. Furthermore, I manually inspect several potential code smell locations in order to assess whether they represent actual problems or not. Finally, I the count the number of problematic code smells and evaluate the quality of the code smell detector.

Research Questions

The case study focuses on the three following questions:

- *RQ1: Is a metrics-based detection algorithm able to find meaningful instances of variability-aware code smells?* Until now, there is no method of detecting the proposed variability-aware code smells in highly-configurable software systems. We contend that the metrics-based detector Skunk is able to find appropriate code that suffers of problems with regards to understandability or changeability.
- *RQ2: Is it possible to give instances found by a variability-aware code-smell detector a ranking, indicating their severity?*
  Skunk calculates severity values for each code smell based on general metrics.

We argue, that we can use these values to sort, and therefore rank, the instances found by Skunk. Thus the higher a code smell is ranked, the more detrimental is its effect on changeability and understandability.

- *RQ3: Does the smells exhibit recurring, higher-level patterns of usage?* With this question, we aim at investigating the reason developers introduced the respective smell. In particular, we want to know whether implementation- or domain-specific characteristics foster the occurrence of such a smell. Moreover, following RQ 1, we investigate which of the pattern are more likely to negatively impact the underlying source code.

The study will prove 1) the existence of the proposed code smells in a wide range of software, and 2) the applicability of the implemented code smell detector. Developers will be able to find potentially harmful variability-aware code smells and eliminate them afterwards. Furthermore, Skunk can be used for future research, for example the development of variability-aware code smells during the evolution of software systems.

Structure

The thesis is structured as follows: In Chapter 2, we give you background information about highly-configurable software systems and code smells. Next in Chapter 3, we explain the concept of variable-aware code smells in such systems. Furthermore, we present the concept of Skunk. We offer a more technical standpoint of Skunk and its implementation in Chapter 4.

Chapter 5 contains all information of our case study. We give you information about setup, the software we examine and the configuration of code smells. We continue with the results of the case study. Additionally, we assess and discuss our findings to answer our research question. We conclude this thesis in Chapter 6. Finally, chapters 7 and 8 describes works that are related to ours and an outlook for further research.

2. Background

The following section will give you background knowledge needed to fully understand the concepts of code smells and highly-configurable software systems. The first part contains details about the definition, problems, variants of code smells. Additionally, we provide an insight in different detection approaches for code smells. The second part consists of information about highly-configurable systems. We will talk about the motivation and advantages of this method. Furthermore, we present the annotation-based approach for implementing such systems.

2.1 Code Smells

For the remainder of this thesis, it is necessary to understand the basics of code smells. Therefore, we discuss what a code smell is and which potential effects they have on source code. Additionally, we present examples of different code smells. Moreover, we examine several approaches of how to automatically detect such smells.

2.1.1 General

There is a large body of research that focuses on refactoring in combination with the definition of code smells and their impact. According to Fowler and Beck, a code smell is a "*surface indication that usually corresponds to a deeper problem in the system*" [Fow99]. Suryanarayana et al. defines that "*[code] smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality*" [SSS14]. Both definitions state that a code smell has a negative effect on software systems and the code design. However, they also add that a code smell is only an indicator or a sign of design flaws of a system. It is not similar to a syntax error that can be found by compilers. Rather, a code smell is bad practice that could pose a problem to you or other developers, but not necessarily.

One of the problems that may arise due to source code suffering of code smell is a decreased program comprehension. Abbess et al. conducted an empirical study of two

code smells [AKGA11]. In their study, they concluded that the appearance of both increases the time it takes to understand code fragments. Additionally, they asked participating developers several questions regarding the source code to be inspected. The occurrence of code smells increased the number of incorrectly answered questions. Other researchers such as Du Bois et al. supported this by stating that code smells, such as Large Classes, negatively effect the comprehensibility of the functionality of a class [DBDV+06]. Another possible difficulty that code smell may pose is the increased maintenance, that is, the time a developer has to invest to repair bad code. Yamashita states that code smells are an indicator for problematic files, where errors are easily propagated [Yam13]. Thus, such problematic files require more bug fixing time than others. Moreover, he concludes that especially the combination of code smells increases this problem. Thummalapenta expressed that error propagation in duplicated code is a problem as it increases the time used for maintenance [TCADP10]. Code smells may pose a problem in the evolution of software systems. In their research, Khomh et al. stated that software with code smells is more change-prone than systems without them [KPG09]. During the evolution of systems, they found that source code with code smells received more changes than others. Olbrich et al. supported this by [OCS10] identifying that large classes are more frequently changed and contained more flaws than other classes.

While not every code smell is a problem, it is necessary to find out why they are introduced. Vale et al. stated that typical factors include short deadlines, time pressure for developers, and bad architectural and design decisions. Moreover, developers might lack the understanding of the business or are not experienced enough. Furthermore, the developers may have a low priority on software quality [VSSA14]. Refactoring has been proposed as a countermeasure for code smells by Fowler and Beck [Fow99]. Code smells can also be avoided by strategies such as additional time for refactoring in the development schedules, higher priority for software quality, better understanding of the adopted technologies and training for inexperienced developers [VSSA14].

2.1.2 Examples

Several researches have proposed typical recurring patterns which are classified as code smell (or anti-pattern). While Fowler focused on refactoring in his work, he also addressed several patterns that indicate of when to refactor [Fow99]. The Long Method is one example of such patterns. This smell is characterized by a method or function that implements to much functionality, that is, a method with a large number of lines of code. The problem is that, the longer a method gets, the harder it is to read, understand and troubleshoot. Changing a code fragment in the early parts of the method may result in unexpected errors later. Splitting the method into smaller methods is a good measure to counter this smell. In Listing 2.1, we can see a code fragment that suffers of this smell. The function starts with line 2835 and ends 2645 lines of code later. We argue that the function can be split into smaller methods to increase understandability and changeability of the source code.

```
2835     static int
2836 win_line(wp, lnum, startrow, endrow, nochange)
2837     win_T   *wp;
2838     linenr_T   lnum;
2839     int      startrow;
2840     int      endrow;
2841     int      nochange UNUSED;  /* not updating for changed text */
2842 {
2843     int      col;         /* visual column on screen */
2844     unsigned   off;         /* offset in ScreenLines/ScreenAttrs */
2845       ...
```

```
5469       ...
5470       #ifdef FEAT_SPELL
5471     /* After an empty line check first word for capital. */
5472     if (*skipwhite(line) == NUL)
5473     {
5474    capcol_lnum = lnum + 1;
5475    cap_col = 0;
5476     }
5477 #endif
5478
5479     return row;
5480 }
```

Listing 2.1: Example of a Long Method. Excerpt taken from VIM screen.c

Another pattern, Fowler presented in his work, is the Duplicated Code or Code Clone anti-pattern [Fow99]. A Code Clone is a sequence of source code that occurs more than once in a software system. The repeated sequence can be an equal clone, but often it is a slightly changed clone. Such changes may include naming of variables, changing types or calls of other methods. The main problem of this smell is an increase of maintenance costs. When a bug is found in a code fragment, it must be fixed. Multiple clones of this code fragment means that there may be other locations where the bug occurs. As a consequence, each clone that suffers of this bug must be fixed too. A developer must spend an increased time in finding and fixing clones with the same bug. An example of this pattern can be seen in Listing 2.2. The code fragment shows two methods that are nearly identical. However, they are used for two different aspects - one for drawing alpha in 24-bit range, the other for drawing alpha in 32-bit range. They differ in the methods they call and their name, but nothing else. This may lead to two problems. First, it may be very error-prone to call one of these methods. A developer may quickly misspell the method name when calling the method. Then the results are unexpected. Each misspelled call must be fixed later. Another problem may occur in one of the `if` statements, if the conditional expression changes in the evolution of the system. While two clones are easy to handle, additional clones constantly increase the time needed for maintenance.

Besides others, Fowler further proposed the Large Class (a class with a large number of lines of code), the Long Parameter List (a method with a large number of input

```
1  void vo_draw_alpha_rgb24(int w,int h, unsigned char* src, unsigned char *↩
       srca, int srcstride, unsigned char* dstbase,int dststride){
2      if(gCpuCaps.hasMMX2)
3          vo_draw_alpha_rgb24_MMX2(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
4      else if(gCpuCaps.has3DNow)
5          vo_draw_alpha_rgb24_3DNow(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
6      else if(gCpuCaps.hasMMX)
7          vo_draw_alpha_rgb24_MMX(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
8      else
9          vo_draw_alpha_rgb24_X86(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
10 }
11
12 void vo_draw_alpha_rgb32(int w,int h, unsigned char* src, unsigned char *↩
       srca, int srcstride, unsigned char* dstbase,int dststride){
13     if(gCpuCaps.hasMMX2)
14         vo_draw_alpha_rgb32_MMX2(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
15     else if(gCpuCaps.has3DNow)
16         vo_draw_alpha_rgb32_3DNow(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
17     else if(gCpuCaps.hasMMX)
18         vo_draw_alpha_rgb32_MMX(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
19     else
20         vo_draw_alpha_rgb32_X86(w, h, src, srca, srcstride, dstbase, ↩
               dststride);
21 }
```

Listing 2.2: Example of a Duplicated Code. Excerpt taken from MPlayer osd.c

parameters) or Speculative Generality (a method or class that may not even be used). Another research on refactoring and code smell patterns by Van Deursen et al., focused on test code [vD+02]. They examined patterns such as Lazy Test (several methods check the same thing) or Test Code Duplication (similar to Duplicated Code in test code). Astels extended this research by adding the Dependent Test Code, Assertion Roulette and Divergent Change smells [Ast03].

2.1.3 Detection

Multiple techniques have been addressed to detect potential code smells. The most basic approach to find code smells is to *manually* analyze source code. Travassos et al. for example, proposed a process based on manual inspection and reading techniques [TSFB99]. This reading techniques give specific and practical guidance of how to detect a smell. While a manual method is possible as an detection approach, it is not feasible for any large scale system.

Another method that extends the manual approach is to *visually* show users specific design flaws. After an automated detection process on a source system, certain characteristics or smells are displayed directly in the source code. This frees the developer of tediously analyzing the source code line per line. He only needs to decide if the detected

code parts are suffering of a smell. Van Emden and Moonen presented such an approach in their work [VEM02]. Here, a code smell is characterized by *code smell aspects*. For example, a code smell aspect can be a statement such as "*method M contains a switch statement*". The proposed detection algorithm finds entities that may suffer from code smells (methods, classes...) and inspects these entities for smell aspects. More complex aspects (e.g., relations between methods and classes) can be derived from so called *primitive aspects*. After the detection process all entities are displayed in a structured graph, where a node is an entity. The edges between entities symbolize relations (e.g, method calls) between them. If one or more smell aspect is found in an entity, the corresponding node can be colorized or changed in size. This gives a developer a quick overview of where to look for design flaws.

More advanced automatic techniques are to detect code smells based on metrics or to use heuristics to automatically decide what a code smell is an what is not. DECOR is a method by Moha et al. [MGDLM10] where a code smell is described using rules based on metrics. A rule consists of a vocabulary often found in description texts of code smells. This vocabulary is then combined with a set of operators defined by DECOR. For example, a LONG METHOD can be described via `RULE:LongMethod {METRIC LOC_METHOD VERY HIGH 10.0};`. This rule describes that a LONG METHOD is defined by a large number of lines of code. More complex code smells can be defined by combining different rules. DECOR uses these rules to generate a detection method. After performing the detection method on a software system, a list of code parts that are potentially suffering of code smells is returned.

Khom et al. proposed a heuristic technique [KVGS09] that employs a Bayesian Belief Network (BBN) [Pea14]. They consider code smell detection as a classification problem, where the decision is whether a code fragment is a code smell or not. The decision is based on multiple vectors that describe both classes. Int this work, they use a BBN to build a model of an anti-pattern and assign a probability to all classes. The output of the BBN is the probability of its output node $p(smell = true)$. The probability p can than be used to sort potential code fragments in order of importance.

2.2 Highly-configurable Software Systems

Today, the market for software development is rapidly evolving. A software development company has to keep its development cycles short to be variable regarding the newest trends. Long development cycles and complex software structures reduce the ability of developers to react to the newest market trends. Therefore, building a product from scratch is not feasible anymore. Optimally, existing products and artifacts should be reused and improved during development of new products. A method to achieve this, is to incorporate the development of *Highly-configurable Software Systems* also known as *Software Product Line*. In the following, we provide the basic knowledge for understanding this development process.

2.2.1 General

A *Highly-configurable Software Systems* is described as a set of program variants that share a common set of features and characteristics. These features are tailored for a specific common market segment or domain. The main goal of software product lines is the reuse of software artifacts [ABKS13][CN02]. A consequence of the reuse is the reduction of maintenance cost, a reduced time to market or the enhancement of quality [PBvdL05]. Such software system is build out of a set of features where one feature represents a specific demand of a stakeholder, a similarity, or a difference between system variants. An example of such software system can be a simple chat system. Typical features may include the logging or encryption of messages, multi-chat rooms or emotes.

In many cases a highly-configurable software system contains multiple features interacting with each other. A typical relation between features is that one feature is dependent on the existence of another feature [ABKS13]. A large number of features and relations complicate the understanding of the system. A common way to structure features is to use a *feature model*. A feature model is a basic abstraction of specific domains and its relationships between features. In Figure 2.1, we show an example of such model. In a feature model, a feature is always shown as a block with a defining name. Here, we have a basic feature called *Base*. This feature is necessary for three other features: *OS*, *Read*, and *Write*. A dependance of a feature is symbolized by a line between two features. Additionally, *Read* and *Write* are optional features indicated by the empty circle. In contrast, the feature *OS* is mandatory for the software system. Moreover, the features *Win* and *Unix* are dependent on *OS* but both can not be active at the same time (indicated by the half-circle). With the help of a feature model, a configuration of to be included features can be chosen where a different set of features creates an other variant of the system.

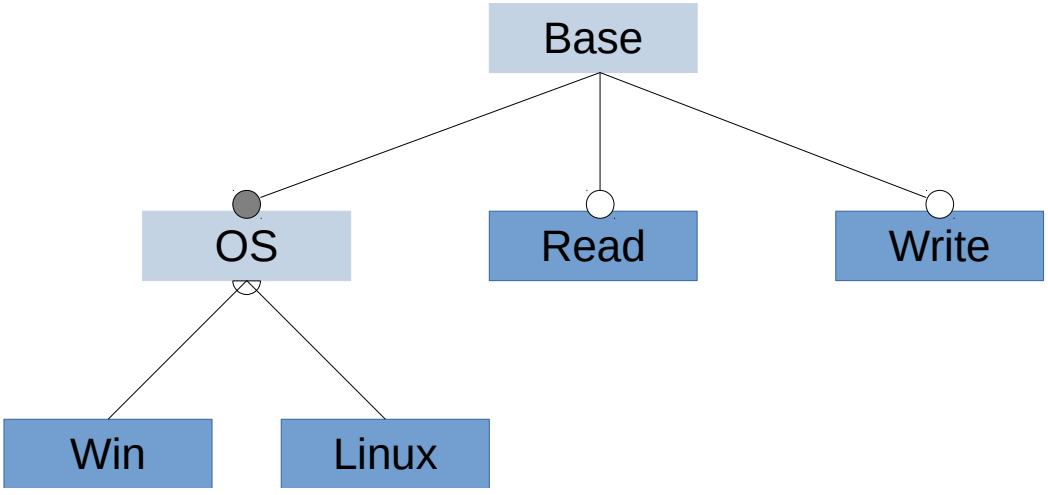

Figure 2.1: Example of a Feature Model

2.2.2 Implementation

In comparison to object-oriented programming, the software suite to compile a highly-configurable software system needs additional functionality to enable variability. One way to do so is to use *preprocessors*. A preprocessor is a program that processes data to generate an output for another program. A common use case in computer programming is a preprocessor that processes source code an generator output for a compiler [ABKS13]. An example is the general-purpose, text-based *C Preprocesser* (CPP) used for the C programming language. Its preprocessing method is dependent on directives such as `#include` for file inclusion or `#define` for macro definition.

Most important for this thesis is the *conditional compilation* of the CPP - a functionality that enables developers to implement variability in software system. On top of that, it is technique that is widely used in academia and industry to implement variability [ABKS13]. The directives of CPP used to conditionally include text are `#if`, `#ifdef`, `#else`, `#elif` and `#else`. In the context of this thesis we refer to each directive simply as `#ifdef`. These directives can be used to annotate features (or the *feature code*. During preprocessing the annotated code is conditionally removed and thus not compiled. Whether or not to remove annotated code is based on a conditional expression of the `#if`, `#ifdef` or `#elif` - the *feature expression*. Code that is annotated by a *true* condition is part of the source code to be compiled. A typical condition is to test whether a name (*feature constant*) is defined as a preprocessor macro. More complex conditionals can be created by combining multiple feature constants using parentheses and logical operators. Moreover, it is possible to use arithmetic tests as conditional expressions.

In Listing 2.3 and following, we present a simple example of the use of CPP directives and their effect on the software system. The purpose of the method `rotate` is to rotate either two or three values. InListing 2.3, we can see the inclusion of three `#ifdef` blocks in line 4, 13 and 18. If the feature `ROTATE_THIRD` is defined as a macro, CPP processes the source code in such a way that the compiler receives the source code of Listing 2.4. Here, the method receives three input values and rotates all values. If the feature is not defined, CPP removes lines 5, 14, 19, 20 and 21 before compilation, as seen in Listing 2.5. In this case the method only takes two parameters and rotates two values. In conclusion, with the implementation of the feature `ROTATE_THIRD`, we receive two variants of the same source code based on the definition of the feature.

The use of `#ifdefs` in source code has often been criticized for being hard to understand and susceptible to the introduction of subtle, hard-to-spot bugs [EBN02] [SC92] [MRG14] [MKR+15]. We discuss these problems in more detail in Section 3.1. Nevertheless, CPP is still a popular choice to introduce variability. On the one hand, introducing and removing features with the help of directives is easy to understand and requires little pre-planning. The `if-else` conditional model is known by every developer and can be directly transferred to the use of `#ifdef` directives. On the other hand, CPP is a part of the C compiler and thus there is no need for extra compilation tools [KA09].

```
1  void rotate
2  (int n1,
3  int n2
4  #ifdef ROTATE_THIRD
5  , int n3
6  #endif
7  )
8  {
9      \\ copy original values
10     int tn1 = n1;
11     int tn2 = n2;
12
13     #ifdef ROTATE_THIRD
14     int tn3 = n3;
15     #endif
16
17     \\ move
18     #ifdef ROTATE_THIRD
19     n1 = tn3;
20     n2 = tn1;
21     n3 = tn2;
22     #else
23     n1 = tn2;
24     n2 = tn1;
25     #endif
26 }
```

Listing 2.3: Example of Preprocessor Usage - Actual Source Code

```
1  void rotate
2  (int n1, int n2, int n3)
3  {
4      \\ copy original values
5      int tn1 = n1;
6      int tn2 = n2;
7      int tn3 = n3;
8
9      \\ move
10     n1 = tn3;
11     n2 = tn1;
12     n3 = tn2;
13 }
```

Listing 2.4: Example of Preprocessor Usage - Compiled with ROTATE_THIRD

```
1  void rotate
2  (int n1, int n2)
3  {
4      \\ copy original values
5      int tn1 = n1;
6      int tn2 = n2;
7
8      \\ move
9      n1 = tn2;
10     n2 = tn1;
11 }
```

Listing 2.5: Example of Preprocessor Usage - Compiled without ROTATE_THIRD

In the context of this thesis we focus on variability implemented by using Cpp directives. Nevertheless, other methods such as *feature-oriented programming* or *aspect-oriented programming* have been in the focus of research [Pre97] [EFB01].

3. Concept

In the last chapter, we discussed the basics of standard object-oriented code smells and introduced the concept of highly-configurable software systems. The definition of object-oriented code smells does not take the variability of highly-configurable systems into account. Therefore, we are unable to neither identify variability-aware code smells nor can we detect those smells using one of the mentioned approaches. In this chapter, we introduce the idea of variability-aware code smells and define three concrete variable-aware code smells. Furthermore, we present Skunk, a new metrics-based code smell detector for these code smells.

3.1 Variability-Aware Code Smells

Highly-configurable software systems are implemented by using variable source code parts. We use `#ifdef` directives to enclose these variable parts in `C` applications. When compiling the software system, CPP removes source code parts that are not part of the current compile configuration. On the positive side, we can use this variability to produce multiple software systems with the same code basis [ABKS13]. However, we always need to use at least two code lines for a variable part [FS15] - one for `#ifdef` and one for `#endif`. While this is easy to comprehend with few `#ifdef` blocks, we need to consider huge software systems with thousands of `#ifdef` blocks. The more variable parts we use, the more obscured the source code gets. In Listing 3.1, we show a simple calculation example with seven different features.

Depending on the configuration, this function adds or multiplies the input with either the default value or another input value. This function can be split into at least four distinct functions. Since not all functions are used in every product, the developers decided to unite the function and add variable features.

On the first look, we observe that the lines of CPP directives are more than the actual lines of C statements. Here, 18 lines of code are used by `#ifdef` directives, while

```
1  int calculate(int a, int b
2  #if defined(USE_C) && !defined(DEFAULT)
3  int c
4  #endif
5  ) {
6  #if defined(DEFAULT)
7     int c = 30;
8  #endif
9    int result =
10 #if defined(DEFAULT)
11   5;
12 #else
13   0
14 #endif
15 #if defined(USE_A)
16  result += a;
17 #endif
18 #if defined(MULTIPLY)
19 #if defined(USE_B)
20    result *= c + b
21 #elif defined(USE_C)
22    result *= c * c
23 #endif
24 #elif defined(ADDITION)
25 #if defined(USE_B)
26    result *= c + b
27 #elif defined(USE_C)
28    result *= c * c
29 #endif
30 #endif
31   return result;
32 }
```

Listing 3.1: Potentially Smelly Annotated Code with Seven Different Features

only 11 lines of code implement actual functionality. Each possible configuration (or distinct function) takes only some of those C statements. For instance, when using feature USE_A, we only have 6 lines of code (lines 1, 5, 9, 13, 16 and 31). Additionally, this function contains nested `#ifdef` blocks from line 18 to 29 and a negation and conjunction in line 2. Feature USE_C activates `int c` in line 3 as input, but not line 28 and 22 as long as feature MULTIPLY or ADDITION are not active. Moreover, line 2 to 4 and 10 to 14 contain undisciplined `#ifdef` blocks i.e. variable code that is smaller than one statement. In total, we can produce 128 variants from this code if the features are not dependent of each other.

A developer has to consider multiple configurations when maintaining or extending this function. Furthermore, he/she has to ensure that this function still works in all possible configurations. In this case, however, the `#ifdef` blocks obscure the code so that it is difficult to quickly understand the function and the relationship with other parts within the software. We conclude that this is a typical characteristic of a code smell.

Fenske and Schulze [FS15] defined new code smells based on object-oriented smells, that take variability into account. In the context of this thesis, we take a look at three specific smells, Large Feature and Annotation Bundle/File.

3.1.1 Large Feature

Similar to the object-oriented smell Large Class, a Large Feature is a feature that is doing too much. Moreover, the feature has too many responsibilities inside the software system. A typical characteristic of such a feature is a high amount of feature code and the introduction of many elements such as variables, methods or classes. In Listing 3.2 we show an excerpt of the feature WIN32 taken from OpenVPN.

```
381 ...
382 #if defined(FEAT_CLIPBOARD) || defined(PROTO)
383
384 static void clip_copy_selection __ARGS((VimClipboard *clip));
385 ...
```

```
1469        ...
1470        #else
1471      return TRUE;
1472 #endif
1473 }
1474
1475 #endif /* FEAT_CLIPBOARD */
1476 ...
```

```
2978        ...
2979        #ifdef FEAT_CLIPBOARD
2980      int           checkfor;
2981      int           did_clip = FALSE;
2982        #endif
2983        ...
```

Listing 3.2: Example of a feature suffering from the LARGE FEATURE smell. Excerpt taken from OpenVPN.

This is an example of a 2569 lines of code long part of the WIN32 feature in the file tun.c. While the task of the feature is easy to identify by its name, it is unclear what specific actions are implemented by this feature. The excerpt in Listing 3.2 alone implements over 50 different functions. Furthermore, the file contains three additional WIN32 feature parts. Altogether, the WIN32 feature covers 6248 lines of code which is equal to 11.2% lines of code of the total software system (55687 lines of code). On top of that, the feature is implemented in 21 files and 173 distinct `#ifdef` blocks, which increases the complexity of the feature even more.

We contend that a feature that is scattered across a high number of files and `#ifdef` blocks is difficult to maintain. A high amount of feature code obscures the functionality of a feature even more, which often results in code that is hard to understand because of the amount of implemented functionality. Furthermore, a slight change can introduce unwanted side effects which makes customization more difficult. Also, some parts may

only be used in conjunction with another feature. Without further examination, a developer may consider some parts as unused. Deletion of this code will create subsequent problems.

3.1.2 Annotation Bundle/File

An Annotation Bundle or Annotation File define the same smell but on a different level of granularity. The Annotation Bundle is a method whose body is for the most part annotated, i.e., it contains a large number of `#ifdef` blocks. On the code level that means that many statements are annotated. Instead of just a few features, the method contains a large number of features. The introducing and `#ifdef` blocks of the feature may even be nested with one another. The Annotation File is a code file with similar characteristics. However, here even classes and global variables can be annotated.

Listing 3.3 shows an excerpt of MySQL version 5.6.17 and bears resemblance to the introducing Listing 3.1. This is a typical example of an Annotation Bundle.

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

Listing 3.3: Example of a function suffering from the ANNOTATION BUNDLE code smell. Excerpt taken from MySQL.

We have 29 lines of code, but only 19 lines are C statements. The remaining 10 lines are made of `#ifdef` directives. Inside these `#ifdef` blocks are nine lines of code and thus only 10 lines of code are base code. This means that half of the code is static while

the rest is dependent on a feature. Furthermore, the function contains five `#ifdef` blocks (6-9, 10-12, 15-19, 21-27 and 22-24) which are controlled by five unique features. On top of that, we have three negations (6, 15 and 22), one nested directive (22-24) and one line of combined features (20).

Based on this, we argue that on the basis of the variable parts, it is - again - hard to understand. It is difficult to know when code is actually used and what is the core functionality of this function. If something is changed, we have to check each configuration which hinders maintenance and evolution.

3.2 SKUNK — Variability-Aware Code Smell Detector

While we have defined variability-aware code smells, we have the problem of identifying them in huge software systems. Current code smell detectors do not take variability into account, and therefore none of the current detectors is able to find them. However, we contend that we can use metrics to describe the characteristics of variability-aware code smells. An Annotation Bundle is problematic due to the amount of `#ifdef` directives within a short range of lines of code, while a Large Feature's typical characteristic is the amount of implement functionality, i.e., lines of feature code.

To this end, we introduce Skunk[1], a metrics-based code smell detection tool for highly-configurable software systems. It uses a straightforward approach to 1) calculate metrics of features, methods and files that take variability into account, and 2) detect possible code-smell candidates by using thresholds and ratios of these metrics. A basic view of the process is shown in Figure 3.1.

Figure 3.1: Skunk- Conceptual Extraction and Analysis Process

The first step is to convert all source code files to XML documents via SourceML [CKM+03]. This step facilitates processing, since we can easily find any `#ifdef` blocks by finding the corresponding texts. Additionally, it is easier to identify which `#endif` directive closes which `#ifdef` block due to the hierarchical nature of XML representation. The converted source code files are not altered by this conversion since we can map all the information back to the standard source code. Next, we use the XML documents as input for Skunk and start a processing step. In this step, we gather all feature-based information on different levels of granularity — feature, method and file. Figure 3.2 shows important terms for this step and Table 3.1 presents the metrics Skunk gathers.

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

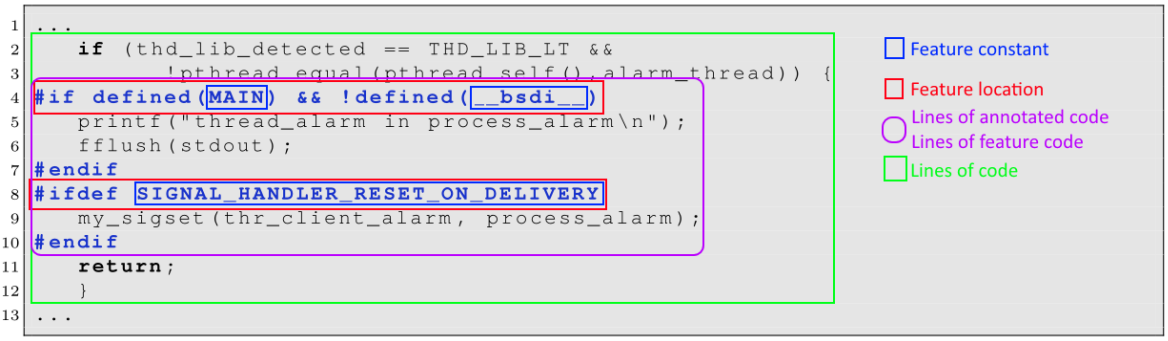

Figure 3.2: Feature Metric Terminology. Excerpt taken from MySQL.

- *Feature constant*: A feature constant is the most basic and smallest unit of a feature. It describes the occurrence of a specific feature inside an `#ifdef` block. Figure 3.2 contains three different feature constants. If we add MAIN to the second `#ifdef` block, the amount of feature constants increases by one.

- *Feature location*: A feature location is the same as an `#ifdef` block. In the example we can count two different feature locations (lines 4 and 8).

- *Lines of annotated code*: An annotated line of code is a line that is enclosed in a feature location. We can see three lines of annotated code (5,6 and 9) in the example in Figure 3.2.

- *Lines of feature code*: Feature code is a line of code that is enclosed by a feature. Each feature location encloses lines of code. Hence, we sometimes count multiple lines more than once. In the example, the feature location in line 4 encloses two lines of code. However, since it contains two different feature constants, we count these two lines for each feature. Therefore, the amount of feature code in this example is five. With the help of this metric, we can easily count the total amount of feature code per feature.

- *Lines of code*: Lines of code is the number of lines written with source code without comments and white spaces.

Skunk assigns these metrics to each feature constant in each feature location. For each feature constant, there is a general feature object and file object to which it is connected to. If the feature constant is part of a method, it is also connected to a method object. The result is a collection of feature constants for each object. Skunk then aggregates the metric values of each object in the collection for the parent object. By this means we get the metrics not only for a feature location, but also for the complete feature, each method and each file. Since this step can take a lot of time in huge software systems, Skunk can save the processed metrics data set to a file. This file can be loaded later to skip the first three steps.

Abbreviation	Metric	Description
LOC	Lines of Code	Total amount of lines with code without comments and white spaces
LOFC	Lines of Feature Code	Total amount of code enclosed by feature constants
LOAC	Lines of Annotated Code	Total amount of code enclosed by feature locations
NOFL	Number of Feature Location	Total amount of different feature locations
NOFC	Number of Feature Constants	Total amount of feature constants without counting a constant twice
NOFC_dup	Number of Feature Constants (duplicated)	Total amount of feature constants with counting doubles
NON	Number of Negations	Total amount of negated feature constants
NOC	Number of Compilation Units	Total amount of source code files
ND	Nesting Depth	The depth of feature constants (top-level constant ND = 0)
NS	Nesting Sum	The amount of feature nesting in a method or file

Table 3.1: Definition of Feature Metrics

In step 4, SKUNK initiates the detection process by loading a code smell configuration file. This file contains multiple customizable ratios and thresholds. The ratios are relationships between the metrics of the feature, method or file objects and the complete project. An example is the ratio between the LOC of the project and the LOFC of a feature. The thresholds define a minimal requirement for certain metrics, for example the minimal amount of NOC or NS.

SKUNK detects potential code smells in the software system by using these ratios and thresholds. It checks each feature constant and its parent objects if they either exceed a threshold or fulfill a ratio. For example, we set the ratio between the LOC of the project and the LOFC of a feature to 0.2. Then, each feature constant of a feature that takes up to at least 20% of the project's LOC gets and indicating attribute for this ratio. Each ratio and threshold has a different attribute. The user can set the ratios and thresholds to mandatory or not. Only feature constants that have at least all mandatory corresponding attributes, will be considered in the output.

The generated output is separated into severity report files and location information files. Both files display all features, methods and files that have fulfilled all mandatory requirements of the code smell configuration file. The severity report displays all basic metrics. Additionally, it contains code smell values that have been aggregated from the basic metric. These values serve as code smell severity indicator. The higher the value, the higher the possibility for a bad code pattern. The user can adjust this value by adding a weighing value to a configuration ratio or threshold. The location information files contain exact location information (file and line) of a potentially smelly source code part. Both files can be used in conjunction for further analysis.

3.2.1 Definition of Severity Values

The severity reports of Skunk contain values indicating the severity of code fragments on understandability and complexity. In the following, we derive formulas for these severity values for each variability-aware code smell from its definition.

We defined a Large Feature as a feature that has a high amount of feature code. Additionally, it introduces many elements such as variables, methods or classes. We derive two relations from these characteristics. First, the higher the `LOFC` of a feature is compared to the `LOC`, the more functionality is introduced by the feature. Second, the more the feature is used (`NOFC`) in feature locations across the project (`NOFL`), the more parts of the source code are annotated by the feature. Combining both leads us to the following formula (ω 1 to 2 indicate the weighing mechanism).

$$LF_{Smell} = \omega_1 * \frac{LOFC}{LOC} + \omega_2 * \frac{NOFC}{NOFL}$$

An Annotation Bundle is a method whose body is for the most part annotated. On the code level that means that many statements are annotated, instead of just a few features, the method contains a large number of features, and `#ifdef` blocks may be nested with one another. We define three characteristics from this definition. First, a highly annotated method body means, that the `LOAC` take up most part of the `LOC` of the method. When this amount of annotated code is introduced by multiple feature locations, the actual code is even more obscured. Second, a small amount of features used in a method is not as confusing as a method with a method consisting of a huge amount of features scattered across multiple feature locations. Therefore, the more features are introduced into the the method, the more unreadable and incomprehensible the code gets. Third, even if the `LOAC` and the `NOFC` of a method is very high, it is not necessarily bad code. A initialization method for example, can initialize a lot of settings per feature — each feature is most likely to be independent of other features. A method with a lot of nested `#ifdef` blocks however, is hard to read and to follow by any developer. Accordingly, a higher `NS` in a method results in more confusing code. We derive the following formula from these three characteristics (ω 1 to 3 indicate the weighing mechanism).

$$AB_{Smell} = \omega_1 * \frac{LOAC}{LOC} * NOFL + \omega_2 * \frac{NOFC_{dup}}{NOFL} + \omega_3 * \frac{ND_{acc}}{NOFL}$$

The Annotation File is similar to the Annotation Bundle, but scoped to files. That means that not only feature occurrences of methods are in the result, but also all other occurrences. Consequently, the metrics of the file-based occurrences are aggregated into the result. Therefore, we get the following formula for calculating this severy value (ω 1 to 3 indicate the weighing mechanism):

$$AF_{Smell} = \omega_1 * \frac{LOAC}{LOC} * NOFL + \omega_2 * \frac{NOFC_{dup}}{NOFL} + \omega_3 * \frac{ND_{acc}}{NOFL}$$

4. Implementation

In this chapter, we present the implementation of Skunk from a more technical perspective. The process diagram for Skunk is shown in Figure 4.1. In the first part, we explain two external tools: SourceML and CppStats, which we took advantage of for the extraction of the metrics. Then, we give a deeper look into the implementation of Skunk. Additionally, we discuss the generation of code smell severity values and the generated output.

4.1 Third-Party Tools

There are existing tools that support the implementation of Skunk. On the one hand we use SourceML, a toolkit for source code to XML transformation, which can be used for structural analysis of source code files. On the other hand we utilize Cppstats, a metrics-based analysis suite for cpp-based variability in software systems. It already measures metrics that Skunk can use to speed up its own detection process.

4.1.1 SourceML

SourceML[1] is a toolkit by Collard and Maletic [CKM+03][CDM+11] for transforming source code into XML documents. It consists of of the document type srcML (Source Markup Language), a code-to-srcML converter src2srcml and a srcML-to-code converter, srcml2src. The conversion can be applied to C, C++ and Java languages. The src2srcML converter enhances the source code of a software system with information from the *abstract syntax tree* (AST) of the source code [PE88] into a XML document per source file. As a result, the source code is preserved with as little processing as possible. In contrast to other C parser, the src2srcml parser does not perform any prepocessing. As a consequence, all cpp directives are preserved. The finished srcML file provides full access to source code lexical, documentary, structural

[1] http://www.srcml.org/

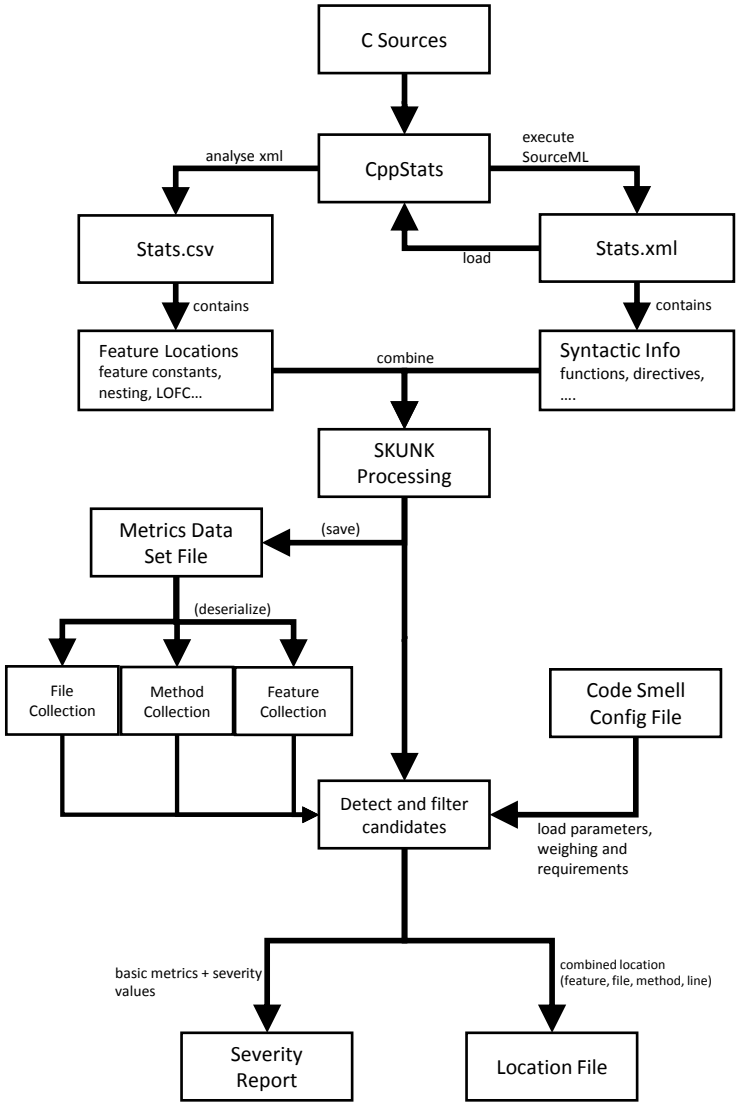

Figure 4.1: Skunk - Data Processing and Smell Detection

and syntactic levels [CDM+11]. Listing 4.2 and Listing 4.1 are an examples of such converted file. shows the original code file.

```
1  #include "rotate.hpp"
2
3  \\ rotate three values
4  void rotate
5  (int n1,
6  int n2,
7  int n3)
8  {
9      \\ copy original values
10     int tn1 = n1,
11     tn2 = n2, tn3 = n3;
12
13     \\ move
14     n1 = tn3;
15     n2 = tn1;
16     n3 = tn2;
17 }
```

Listing 4.1: SourceML - Example of a source code file in the original code.

Every srcML file starts with the XML declaration. Line 2 and 3 contain the `unit` element, which is the parent node for the entire source code file. The `xmlns` and `xmlns:cpp` describe namespaces that are used for the most basic srcML set elements (for instance, functions, statements and parameters), while the latter contains namespaces for the cpp language such as pre-processor directives. The `language` attribute defines the original language and is used for XML to code conversion. The `filename` attribute contains the path to the original source file. Those descriptive lines are the only lines not part of the original source file. The child nodes of then `unit` element contain the source code. In this example, it is a function that rotates the value of three variables. A complete list of srcML element is given in Figure 4.2.

4.1.2 CppStats

CppStats[2] by Jörg Liebig [LAL+10] is a software suite of analysis tools for measuring cpp-based variability in highly-configurable software systems, giving insights about the use and granularity of cpp annotations. It traverses through srcML files to gather variability metrics, such as the number of feature constants, lines of feature code, nesting and tangling degree, or granularity per feature constant on a on a file level.

An enhanced version[3] has been developed for another study by Hunsen et al. [HZS+15]. Besides the original operating mode, it offers multiple processing modes, each gathering a different kind of data:

- `general`: The original CppStats result file with an overview of variability metrics.

[2] http://fosd.de/cppstats
[3] http://www.fosd.net/oss_vs_is

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.sdml.info/srcML/src" xmlns:cpp="http://www.sdml.info/srcML/cpp" language="C++"
filename="rotate.cpp">
<cpp:include>#<cpp:directive>include</cpp:directive> <cpp:file>"rotate.hpp"</cpp:file>
</cpp:include>
<comment type="line">\/\/ rotate three values</comment>
<function><type>void</type> <name>rotate</name>
<formal-params>(<param><type>int&amp;</type> <name>n1</name></param>,
<param><type>int&amp;</type> <name>n2</name></param>,
<param><type>int&amp;</type> <name>n3</name></param>)</formal-params>
<block>{
 <comment type="line">\/\/ copy original values</comment>
 <decl-stmt><decl><type>int</type> <name>tn1</name> = <name>n1</name>,
<name>tn2</name> = <name>n2</name>, <name>tn3</name> = <name>n3</name></decl>;
</decl-stmt>
 <comment type="line">\/\/ move</comment>
 <expr-stmt><expr><name>n1</name> = <name>tn3</name></expr>;</expr-stmt>
 <expr-stmt><expr><name>n2</name> = <name>tn1</name></expr>;</expr-stmt>
 <expr-stmt><expr><name>n3</name> = <name>tn2</name></expr>;</expr-stmt>
}</block></function>
</unit>
```

Listing 4.2: SourceML - Example of a source code file in the srcML format and the original code.

- `generalvalues`: An overview of scattering, tangling and nesting per file.
- `featurelocations`: An overview of each feature location. Each feature location is displayed with the path to the file, the start line, the type of annotation, each feature constants and conditional operators.
- `discipline`: A list that shows the count of annotations per discipline class. A disciplined annotation is an annotation that contains complete code fragments such as the construct of a loop and its executing block. An undisciplined annotation contains code fragments such as parts of an if expression, thus breaking the standard structure.
- `derivative`: A list of all annotations that interact with others. These listed annotations share code with each other in certain locations (annotations with && operator).
- `interaction`: An overview of pair-wise interactions of feature constants that have been used together in one expression.

The enhanced version of CppStats accomplishes two tasks for Skunk. First, CppStats operates on SrcML files and thus needs to execute SrcML before its own analyzing steps. Second, it gathers data about each feature and its feature locations, which we can use this data to skip finding each single feature location in the software

Category	srcML Elements
File/Project	unit
Statement	if, then, else, while, do, switch, case, default, for, init, incr, condition, break, continue, comment, name, type, block, index, expr_stmt, expr, decl_stmt, decl, init, goto, label, typedef, asm, macro, enum, empty_stmt, namespace, template, using, extern
Function/Method	function, function_decl, specifier, return, call, parameter_list, param, argument_list, argument
Class	class, class_decl, public, private, protected, member_list, constructor, constructor_decl, destructor, destructor_decl, super, friend
Struct and Union	struct, struct_decl, union, union_decl
Exception	try, catch, throw, throws
C-Preprocessor	cpp:directive, cpp:file, cpp:include, cpp:define, cpp:undef, cpp:line, cpp:if, cpp:ifdef, cpp:ifndef, cpp:else, cpp:elif, cpp:endif, cpp:then, cpp:pragma, cpp:error
Java	extends, implements, import, package
Extra Markup	literal, operator, modifier
Misc	escape

Figure 4.2: SourceML - List of srcML elements organized in categories [CDM+11]

system. As a consequence, we utilize this additional information to speed up our detection process and work with srcML files without running both tools independently.

4.2 Processing Source Files

Before the actual detection process starts, Skunk processes the source code of the software system into a storeable, reusable and structured data type. Figure 4.3 presents the process from the given input up until the the completion of the metrics data set.

At the start of this phase, skunk utilizes the preliminary work of CppStats. At present, the tool offers multiple analysis options. We only need the results of the `general` and the `featureLocation` option. The generated CSV files of CppStats give information about files that contain features and the location of each `#ifdef` block. Additionally, each `#ifdef` block information already contains details such as the feature constants used in this location and negation status per constant. As a result, Skunk is able to skip files with no feature information, which increases the performance of the process. By using the CSV files, Skunk creates its basic data (feature- and file collections) from which the metrics on the feature-level are derived (`NOFL`, `NOFC`, `NOFC_dup`, `NON`, `NOC ND`, `NS`). Furthermore, the CSV file generated by the `general` option contains the `LOC` of the project. Hence, there is no need to process the complete system.

During CppStats' run-time, SourceML is executed and the XML files of the source system are stored in the CppStats output folder. The resulting source code XML files

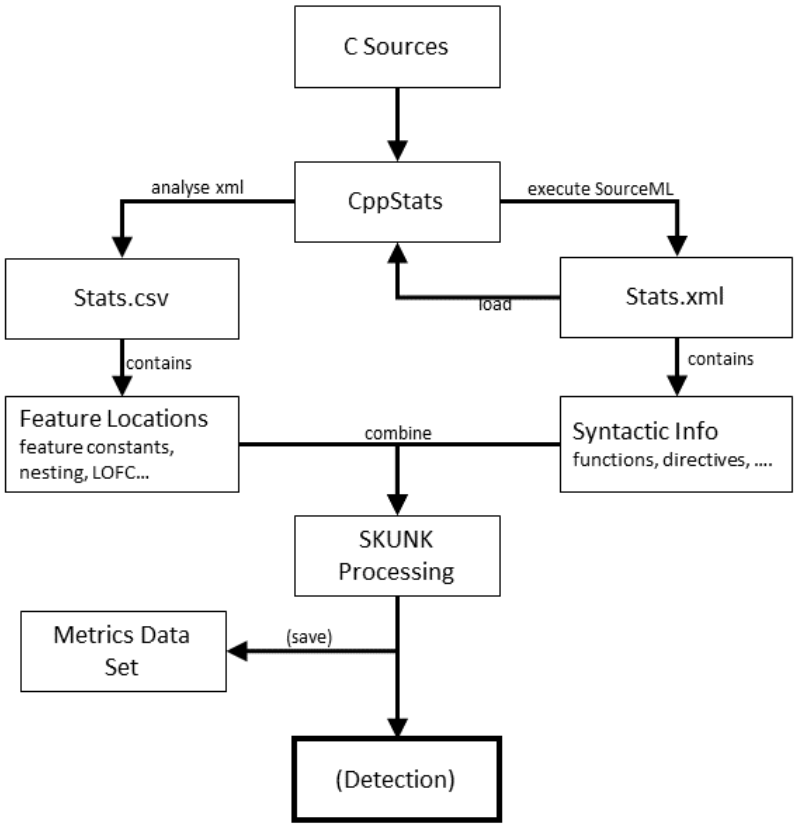

Figure 4.3: Skunk - Data Processing and Metrics Calculation

are used by Skunk for further analysis. At this point, Skunk already knows the location of each feature constant, thus no file needs to be fully processed from beginning to start. The main task of processing SourceML files is to connect feature constants with their corresponding (if available) method and compilation unit. Furthermore, Skunk gathers structural details not covered by the CppStats CSV files. One such detail is, for example, to get the LOFC and LOAC without whitespaces and comment lines. Additionally, most metrics are used on all granularity levels. Hence, when connecting a feature constant with a method or compilation file, the metrics of the feature constant are accumulated into the method, e.g adding the LOAC of the feature constant to the total LOAC of the feature or increasing the ND of the method.

Finally, the complete metrics data set is given to the actual detection process. Before starting the detection process, the metrics data set can be serialized and saved to the hard-drive, if the user specified to do so. Since the computation time of the Skunk processing step is related to the size of the source system, the process may take a lot of time. The data processing and metrics calculation can be skipped by loading a previously serialized metrics data set.

4.3 Detection and Output

In the processing step, Skunk created a metrics data set on which the detection operates. The code smell to be detected is specified in a configuration file. Figure 4.4 presents the steps of the detection process up to the final output.

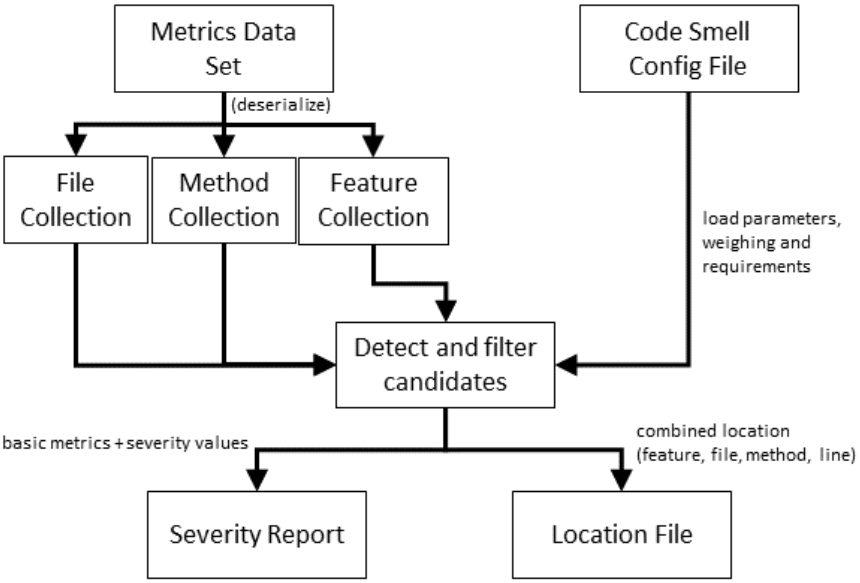

Figure 4.4: Skunk - Detection and Output

First, the metrics collection are loaded and deserialized, if a metrics set from the hard-drive is loaded. Otherwise, the metrics collection is already present due to the prior

processing step. SKUNK uses this data for the detection, while a code smell configuration file contains the details of the code smell to be detected. This file contains multiple thresholds and ratios, which the user can use to describe the code smell. Table 4.1 presents the available configuration parameters. Using a threshold, for example `Feature_NumberNofc`= 2, all features with at least two feature constants are considered smelly. A set ratio on the other hand, considers all elements smelly that exceed the ratio. The ratio `Feature_ProjectLocRatio` with a value of 0.5 for example, requires smelly features to contain at least half of the source systems lines of code.

Name	Definition
`Feature_ProjectLocRatio`	Ratio between LOFC of one feature to the LOC of the system
`Feature_MeanLofcRatio`	Ratio between LOFC of one feature to the mean LOFC of all features
`Feature_NumberLofc`	Threshold for minimum amount of LOFC
`Feature_NumberNofc`	Threshold for minimum amount of NOFC
`Feature_NoFeatureConstantsRatio`	Ratio between the NOFC_dup of one feature to the total NOFC_dup
`Feature_NumberOfCompilUnits`	Threshold for minimum amount of NOC
`Method_LofcToLocRatio`	Ratio between of LOFC in a method to the total LOC in a method
`Method_LoacToLocRatio`	Ratio between of LOAC in a method to the total LOC in a method
`Method_NumberOfFeatureConstants`	Threshold for minimum amount of NOFC in a method
`Method_NumberOfFeatureConstantsNonDup`	Threshold for minimum amount of NOFC_NonDup in a method
`Method_NumberOfFeatureOLocations`	Threshold for minimum amount of NOFL in a method
`Method_NestingSum`	Threshold for minimum amount of NS in a method
`Method_NestingDepthMin`	Threshold for minimum amount of ND in a method
`Method_NegationCount`	Threshold for minimum amount of NON in a method
`File_`	File configurations are the same as for methods

Table 4.1: Code Smell Configuration: Ratios and Thresholds

During detection, SKUNK checks if a ratio or threshold applies to a feature, method or file. If a parameter is exceeded, all occurring feature locations in the specific scope (all locations in a method or file, or all locations of a feature) are considered smelly for the reason of the set parameter.

In addition to the configuration parameters, the user is able to define the parameters as mandatory. A location that does not exceed a parameter are filtered out, and therefore are not part of the result. Thus, feature locations must exceed all mandatory parameters, but can also exceed optional ones. For example: The configuration file contains the configuration parameters `Method_NumberOfFeatureConstants=2` and `Method_LoacToLocRatio=0.1`. Therefore, the result will contain feature locations of methods with either at least two feature constants or at least ten percent of annotated code, or both. If `Method_NumberOfFeatureConstants=2` is set to mandatory, the result contains feature locations of methods with at least two feature constants. These methods can also contain at least two feature constants. If both configuration parameters are set to mandatory, only feature locations with both attributes are part of the result.

Besides the mandatory setting, a configuration parameters can be weighed. The weighing mechanism is used in one of the output files, in such a way, that the results can be tuned towards a parameter. The higher a parameter is weighed, the higher is the impact of this parameter.

When the detection and filtering finished, SKUNK generates two output files for each scope, the severity reports and the location files, which we explain in the following sections.

4.3.1 Severity Reports

A severity report consist of a sorted list of features, methods or files. Each object is connected with smell severity values and their underlying basic metrics. The smell severity values indicate the "smelliness" of an object. We contend that, the higher the value is, the higher is the possibility of an actual code smell. At present, we implemented smell severity values for the LARGE FEATURE and the ANNOTATION BUNDLE/FILE. SKUNK creates one severity report per granularity level.

The *feature severity report* displays the name of the feature and all necessary metrics for identifying LARGE FEATURES. That includes the following basic metrics: the `LOC` of the source system and the `LOFC`, `NOFC` and `NOFL` of the feature. Furthermore, it includes the sub-results of the $LFSmell$ formula. First, $LOFC_{Smell}$ indicates the result of the first part ($\omega_1 * \frac{LOFC}{LOC}$). ω_1 can be set with the weighing of `Feature_NumberLofc`. Second, `ConstantsSmell` shows the result of the second half of the formula ($\omega_2 * \frac{NOFC}{NOFL}$). Here, ω_2 is set by the weighing of `Feature_NumberNofc`. `LFSmell` is the sum of both sub-results.

Besides, the file, name and starting line, The *method severity report* shows all metrics for identifying ANNOTATION BUNDLES: `LOC`, `LOAC`, `NOFL`, `NOFCdup` and `NDacc`. The sub-results of the AB_{Smell} are displayed as $LocationSmell$ ($\omega_1 * \frac{LOAC}{LOC} * NOFL$), $ConstantsSmell$ ($\omega_2 * \frac{NOFC_{dup}}{NOFL}$) and $NestingSmell$ ($\omega_3 * \frac{ND_{acc}}{NOFL}$). Here, ω_1 is set by `Method_LoacToLocRatio`, ω_2 by `Method_NumberOfFeatureConstants`, and ω_3 by `Method_NestingSum`.

The *file severity report* is built up similar to the *method severity report*, but with values scoped to files.

4.3.2 Location files

While the severity reports give a quick overview of potential methods, files or features affected by code smells, it is troublesome to find the participating code section in the source system. Therefore, Skunk generates files that display the exact location of each detected smell. At the start of each file, the code smell configuration parameters of the detection process are displayed. The main part consists of line information per feature constant sorted after the corresponding scope.

In Figure 4.5, we show an excerpt of a location file for feature scope. In this case, the user was looking a Large Feature. The feature `ENABLE_CRYPTO_OPENSSL` is spread across 4 files (line 115, 118, 121, 124). Each of those four files contain a feature location with the feature `ENABLE_CRYPTO_OPENSSL` (line 117, 120, 123 and 126). A list of smell reasons is displayed after the location of the feature.

```
114 [Feature: ENABLE_CRYPTO_OPENSSL]
115 File: /home/dwelgaz/Desktop/Development/Thesis/Software/cppstats/sourcecode/_cppstats_featurelocations/src/openvpn/crypto_openssl.c.xml
116 Start       End     Reason
117 38      785     [LARGEFEATURE_LOFCTOMEANLOFC]
118 File: /home/dwelgaz/Desktop/Development/Thesis/Software/cppstats/sourcecode/_cppstats_featurelocations/src/openvpn/pkcs11_openssl.c.xml
119 Start       End     Reason
120 38      192     [LARGEFEATURE_LOFCTOMEANLOFC]
121 File: /home/dwelgaz/Desktop/Development/Thesis/Software/cppstats/sourcecode/_cppstats_featurelocations/src/openvpn/ssl_openssl.c.xml
122 Start       End     Reason
123 38      1372        [LARGEFEATURE_LOFCTOMEANLOFC]
124 File: /home/dwelgaz/Desktop/Development/Thesis/Software/cppstats/sourcecode/_cppstats_featurelocations/src/openvpn/ssl_verify_openssl.c.xml
125 Start       End     Reason
126 38      628     [LARGEFEATURE_LOFCTOMEANLOFC]
```

Figure 4.5: Skunk - Excerpt of a location file scoped to features

5. Evaluation

In this chapter, we present a case study to investigate the existence of the discussed variability-aware code smells. Moreover, we demonstrate the effectiveness of the detection algorithm of Skunk. To this end, we formulate our objectives, explain the subject systems used in the case study and discuss our methodology, which is based on our recent paper [FSMS15]. Furthermore, we presents the results and assess the precision of Skunk by manually inspecting potential Annotation Bundles and Annotation Files. We use a statistical approach for deciding, when a feature is considered to be a Large Feature. Finally, we discuss the results regarding our research questions provided in Chapter 1.

5.1 Objectives

In this study, we focus on three variability-aware code smells that we want to detect in multiple subject systems of different domains. We basically want to evaluate the usefulness of both, our concept of variability-aware code smells, as well as our method to detect them. Therefore, we focus on three objectives

First, we want to show that a metrics-based detection algorithm is able to find meaningful instances of the proposed smells. For the Annotation Bundle/File, we achieve this by detecting potential smelly code fragments in multiple source systems with Skunk. The detected code fragments then need to be manually inspected by the author, since there is no baseline for variability-aware code smells up to this point. During this inspection, the author decides if the code fragments suffer of problems with regards to understandability and changeability.

A manual inspection for a Large Feature at this point is not possible. With the Large Feature detection, we want to features that implement an excessive amount of functionality. However, without deep knowledge of the subjects systems it is very difficulty to objectively decide what *is* an excessive amount. Therefore, we consider

the LOFC of the features as an indicator. The more lines of code the feature implements, the more functionality is implemented by the feature. The more functionality is implemented, the longer a developer needs to fully understand a feature. Abbes et al. have examined this problem with the object-oriented smell Large Class (or Blob in their work) [AKGA11]. Therefore, we want to find thresholds at which point features implement an excessive amount in comparison to "normal" features. We calculate these thresholds using a statistical approach. The maximum thresholds define at which point a feature is an outlier. An outlier is considered to be a Large Feature.

Second, we want to investigate, if the proposed severity values in Chapter 3, prove to be a good indicator for the "smelliness" of a code fragment. We contend, that higher values indicate a more negative effect on the understandability of source code, thus enabling us to rank detected fragments after the calculated severity values. To prove this, we have to compare a set of high-ranked code fragments to a set of low-ranked ones in terms of precision. The precision of Skunk per set increases with each code fragment, that has been identified as actual code smell during the inspection. The severity values a good indicator, if the precision for the high-ranked set is higher than the precision for the low-ranked set. For the Large Feature, we examine the distribution of actual feature sizes. We use the severity value of the Large Feature to rank all features of all systems. We contend, that the higher a feature is ranked, the larger they are than lower-ranked features. Furthermore, only higher ranked features will exceed the thresholds we calculate.

Third, we investigate if the detected smells exhibit recurring, higher-level patterns of usage. Therefore, we check during the manual inspection, if there are multiple fragments that are considered to be smelly because of the same (or similar) reason.

5.2 Subject Systems

To perform the case study, we need subject systems which Skunk can process to detect possible code smells. For this purpose, we require software systems that implemented its variability by using cpp directives. Furthermore, the systems source code needs to be freely available.

Moreover, the subject systems need to fulfill the following requirements:

- *Medium size:* To get a sufficient number of potential variability-aware code smell instances, the systems need to have a considerable amount of lines of code. However, software systems with a massive amount of lines of code such as Linux, are infeasible to manually inspect. Therefore, we need to show the applicability of Skunk to systems of medium size.
- *Significant amount of LOAC:* Software systems with a higher amount of annotated code are more likely to contain variability-aware code smells.
- *Different domains:* The software systems need to be of different domains, to remove bias regarding specific properties of a particular application domain.

To this end, we chose a subset of the forty systems used by Liebig et al. to analyze preprocessor usage [LAL+10]. Considering our requirements, we selected the software systems presented in Table 5.1. A graphical representation can be seen in Figure 5.1 and Figure 5.2. For each system, we list the version, its domain, the lines of code and the percentage of annotated code to all code. Additionally, we include the number of features, the number of feature locations, and the amount of feature constants. The LOC (only non-blank and non-comment lines of C code) have been measured by the CLOC[1] tool, while the %LOAC, #CU and NOFC are part of SKUNK's metrics.

Name	Version	Domain	LOC	%LOAC	#Feat	NOFL	NOFC
BUSYBOX	1.23.2	UNIX utility collection	182,555	25.6	1263	4435	5129
EMACS	24.5	Text editor	247,403	29.0	353	833	1140
LIBXML2	2.9.2	XML processing library	215,751	69.5	406	4963	5735
LYNX	2.8.8	Text-based web browser	115,102	43.8	519	3587	4150
MPLAYER	1.1.1	Media player	673,388	9.1	1079	4868	6549
PHP	5.6.11	Scripting Language	117,813	18.2	556	2569	3586
VIM	7.4	Text editor	285,817	69.8	841	13413	16428

Table 5.1: Overview of the chosen subject systems

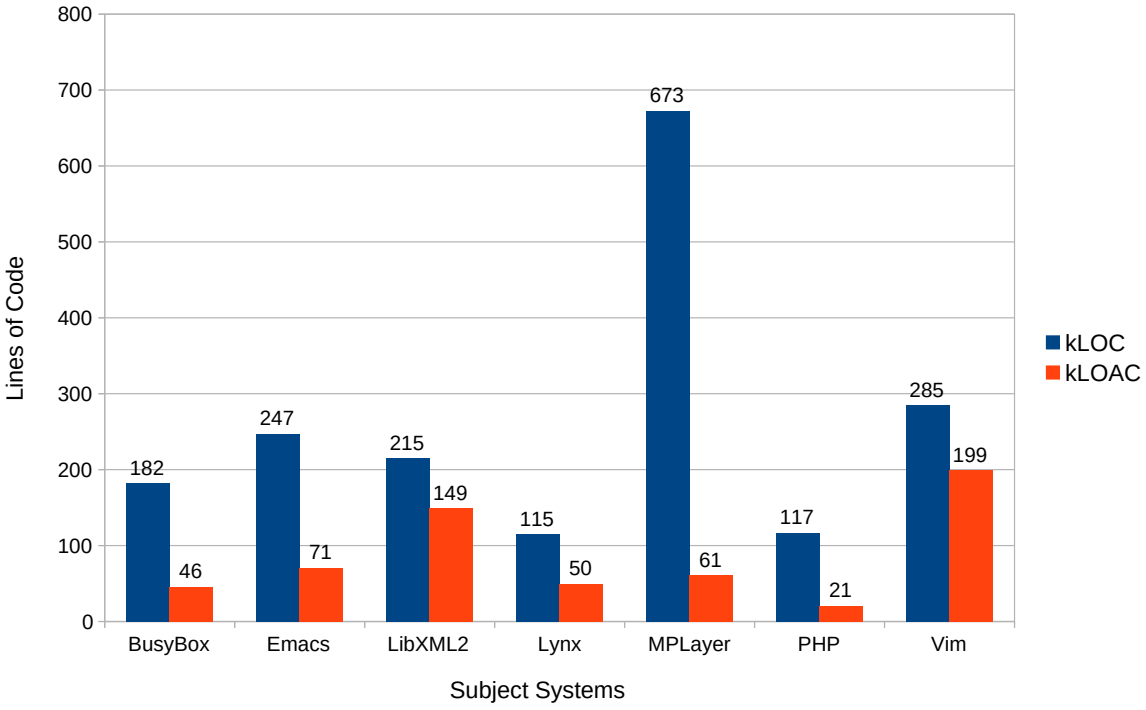

Figure 5.1: Comparison of the subjects systems lines of code to lines of annotated code

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

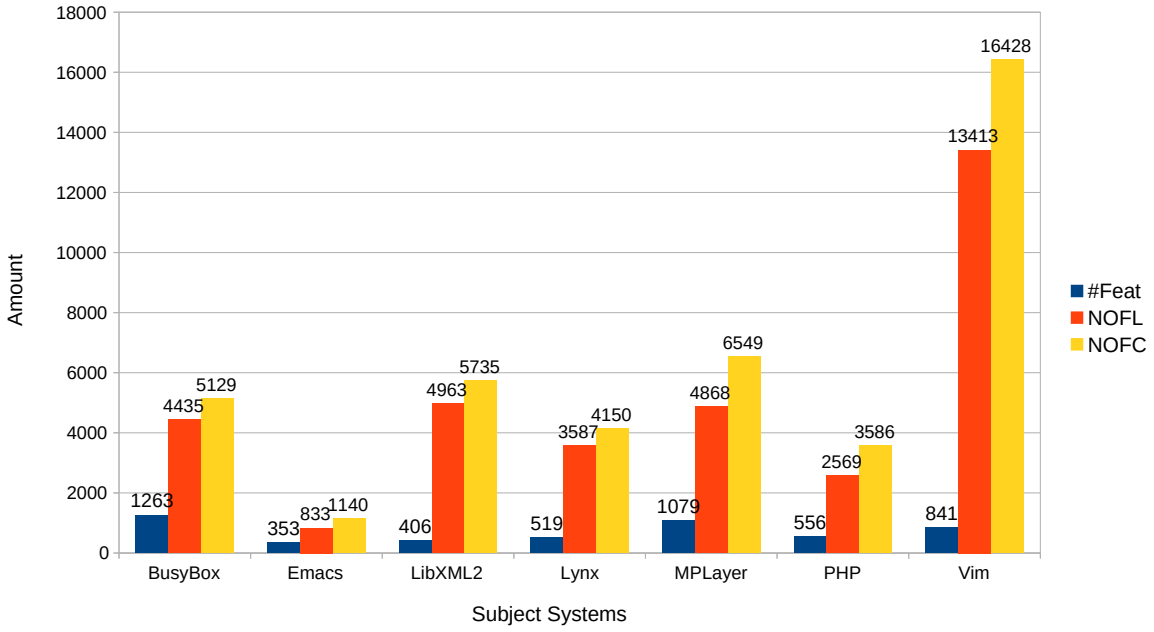

Figure 5.2: Comparison of the subject systems number of features to the number of feature locations and number of feature constants

5.3 Methodology

In the following, we present the method of how we conduct the case study. In parallel to this thesis, we published a paper [FSMS15], which this method is based upon. Here, the authors validated the results for the Annotation Bundle. In the present work, I evaluate the variability-aware code smell Annotation Bundle, Annotation File and Large Feature with the following steps: setting default parameters, creating a sample set and validating the samples. We describe these steps in the following sections.

5.3.1 Setting Default Parameters

As mentioned, the user has the ability to define thresholds and ratios for a code smell in Skunk. In Chapter 4, we already defined the necessary configuration parameters for the variability-aware code smells we want to detect. For this case study, we decided to use the following values:

- Annotation Bundle:
  LOAC $ratio = 0.5$: The smell refers to functions with problematic usage of directives, and therefore we require each function to contain at least 50% annotated code. A lower threshold may lead to too many false-positives.
  NOFC $_{dup} = 2$: We require the occurrence of at least two different features in the function, since we consider scattering and tangling in the AB_{Smell} function.
  NOFL $= 2$: To avoid to get functions with only one large section of annotated

code introduced by one feature location, we require the function to contain at least two different feature locations.
`ND` = 1: We argue that nesting of feature locations highly increases the complexity of this smell. Therefore, we only want to include functions with at least on nested `#ifdef` block.

- Annotation File:
  `LOAC` *ratio* = 0.5: A file with a huge amount of annotated code - possibly introducing new methods and parameters - is most likely to be hard to understand and maintain. Thus, we require files to contain at least 50% annotated code.
  `NOFC` $_{dup}$ = 2: We require occurrence of at least two different features in the file, since we consider scattering and tangling in the AF_{Smell} function. One feature alone, would have less effect on the source codes readability, since only one variant with the feature code, and one variant with the feature code would exist. More features, especially in complex and nested feature locations, increases the complexity of source code significantly.
  `NOFL` = 2: To avoid to get files with only one large feature location, we require the file to contain at least two different feature locations.
  `ND` = 1: We argue that nesting of feature locations increases the complexity of this smell. Therefore, we only want to include files with at least one nested `#ifdef` block.

- Large Feature:
  For the statistical analysis, we need the information and metrics of all features. Therefore, we set `LOFC` 1 By this means, Skunk includes all features and necessary metrics in its output files. We use the severity files for statistical calculations.

Additionally, we set all parameters to mandatory. That means Skunk only reports code elements (features, methods and files) that fulfill all requirements set by the parameters. To reduce the number of likely false-positives, we chose the parameters based on runs of Skunk on other systems. With increased thresholds, Skunk may miss potentially smelly code elements due to a decreased recall.

5.3.2 Sample Selection

We run Skunk with the defined code smell configurations. Based on the parameters, the report files will include all code parts that are potentially smelly. However, it is unclear to what extent the reported code smell instances are false-positives or not. As of right now there is no baseline for variability-aware code smells, and therefore we need to manually inspect a subset of the reported code smells. Inspecting all reported code smell instances is infeasible in a reasonable time. Therefore, we review 20 smells for the Annotation Bundle and Annotation File code smell in each subject system.

Half of the samples are taken from the reported smells with the highest severity value. For the second half of the the samples, we split the remaining result set into 10 equally distributed segments. Then, we select one entry of each segment and add it to the sample set to be reviewed. Through this method, we get a cross-section of all smells for the evaluation. Moreover, we can test if the severity values are a good indicator for a variability-aware code smell or not by comparing the precision of both sample halves.

For example, Skunk reported 66 potential functions inside the source code of the tool MPlayer are suffering of the Annotation Bundle smell. We take the 10 most smelly functions based on the "AB_{Smell}" value for the first half of the samples set. The remaining 56 functions are split into 10 subsets of similar size of 5 to 6 functions per subset. Then, we randomly choose one candidate of each subset and add it to the samples set. Thus, we have a sample set of 20 potentially smelly functions that we need to manually review. The sample set is then manually inspected by the author in the next step.

The sample size for the Large Feature statistical analysis will be the total amount of features of all subject systems.

5.3.3 Validating the Samples

We use the previously created sample set for assessing the precision of our detection algorithm. For this purpose, we assign a rating to each reviewed sample, thus, simulating a human oracle. We chose a 3-point rating from -1 to 1. Samples with a rating of -1 are samples with no negative impact on the source code. These are false-positives in the reported code smells. A rating of 0 describes samples with partial negative impact, for instance, a sample where only certain code fragments have a negative impact on understandability or changeability, but the rest of the code does not suffer from variability-related problems. The rating 1 indicates samples where most of the sample's source code negatively impacts the code. In our recent paper [FSMS15], Fenkse and Schulze evaluated the samples independently and combined their results afterward. Here, only the author evaluates the samples.

In the evaluation, we focus on how our smells increase complexity and readability of the reported code sections. Furthermore, we try to identify the location for a change, if possible. Based on the review, each reported smell is assigned a rating. The author records the reason of why a sample constitutes a smell or not and the functionality provided by the sample.

In the statistical analysis for the Large Feature, we want to find outliers considering the `LOFC` and `NOFC` of all features. For this purpose, we use the calculation of three percentiles - 75th, 95th, and 97.5th. The 75th percentile for example, returns a value above which only 25% of all the data is. We choose the threshold, based on the increase of the value from one percentile to another and the number of features that are above this value. For example: The 75th percentile may return a `LOFC` threshold of 100. Above this value are 1000 features with a higher `LOFC`. With the 95th percentile we

get a threshold of 150 and only 500 features are above this value. While this the number of features decreased by 500, the threshold increased by only 50. That means that there are still a lot of features within this range. The result of the 97.5th percentile may return a threshold of 1000. Above this value are 250 features with a higher LOFC. Within only 2.5%, there is a huge increase in the threshold. We consider this to be a sign of uncommonly large features. By this method, we get the limit values for LOFC and NOFC. Only 2.5% of all features are either above the LOFC limit or the NOFC limit. We consider features that are above both limits to be Large Features. We contend that a feature with lots of code scattered across many locations is even harder to understand than a feature with only a few locations.

5.4 Results

In this section, we present the results of our manual inspection. We present the results for each variability-aware smell in a table, and show a comparison of the precision of the top 10 samples with the random 10 samples in a graphical way. Furthermore, we discuss why samples got a specific rating with exemplary code samples. At the end, we focus on noteworthy findings and often occurring patterns within the samples set.

5.4.1 Annotation Bundle

The results for the manual inspection process of potential Annotation Bundles are shown in Table 5.2. As shown, each subject system exhibits potential code smells. However, the amount of detected smells differs considerably. Skunk only found 20 potential code smells in Emacs and 26 smells in PHP. Vim in contrast, exhibits 256 potential Annotation Bundles.

Name	#Pot	AB_{-1}	AB_0	AB_1	AB_{0+1}	% Precision
BusyBox	40	2 (7)	4 (2)	4 (1)	8 (3)	80 (30)
Emacs	20	4 (10)	4 (0)	2 (0)	6 (0)	60 (0)
Libxml2	76	4 (10)	2 (0)	4 (0)	6 (0)	60 (0)
Lynx	76	4 (7)	2 (3)	4 (0)	6 (3)	60 (30)
MPlayer	66	10 (5)	0 (1)	0 (4)	0 (5)	0 (50)
PHP	26	8 (8)	2 (1)	0 (1)	2 (2)	20 (20)
Vim	259	0 (4)	6 (6)	4 (0)	10 (6)	100 (60)
Total	563	32 (51)	20 (13)	18 (6)	38 (19)	54.2 (27.1)

Table 5.2: Results of the manual inspection for the Annotation Bundle Smell, showing the number of potential smells, the ratings and the calculated precision. The first values indicate the top 10 samples, the values in parenthesis indicate the results for the random samples.

In Figure 5.3, we present a comparison of rated samples for the Top 10 samples on the left and the 10 lower rated random samples. In combination with the comparison in

Figure 5.4, we can determine a clear decline in the precision for most subject systems. eMacs and LibXML2 has a 60% higher precision for the top 10 samples, since none of the reported potential smells below the top 10 were considered as negative in terms of understandability. Vim and BusyBox exhibit the highest amount of actual code smells, and therefore Skunk has a higher precision. Nevertheless, the precision for PHP does not differ. Furthermore, none of the top 10 samples of MPlayer were considered to have a negative impact on understandability and complexity. In total, Skunk's precision for the top 10 samples were 27.1 % higher than the precision for the other half of the samples. From this, we conclude that the AB_{Smell} severity rating is, a good indicator for ranking potential Annotation Bundles. The higher the rating, the higher is the possibility for smelly code fragments.

The results of this inspection aligns with the findings of our recent paper[FSMS15], and therefore validates these findings. The precision differs slightly, but we have to consider that the ratings have been decided by humans. Thus, the impression of bad code may differ. However, the overall trend of precision decline with lower severity ratings is the same.

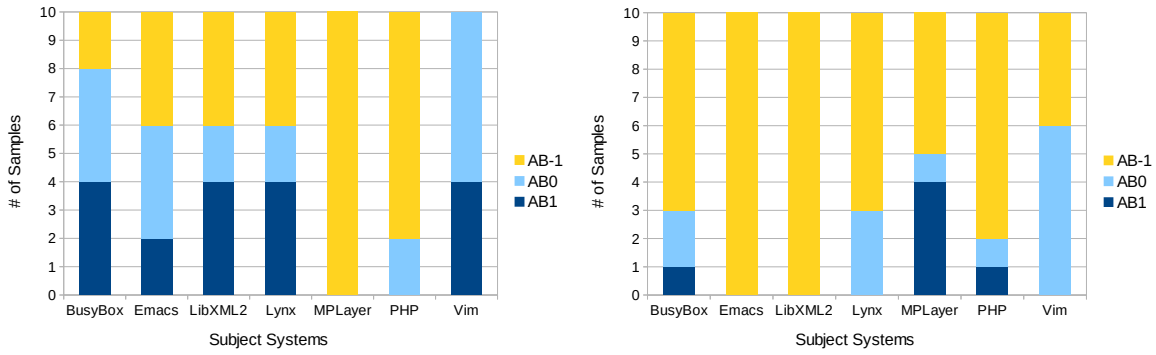

Figure 5.3: Graphical representation of the results of the manual inspection for the Annotation Bundle Smell (Left - Top 10 samples, Right - Random 10 samples).

Now, we show examples of *why* certain samples were given their specific rating. The method `win_line` in the source file `screen.c` was given a rating of 1, meaning that most of the methods source code is negatively impacted with regards to readability and understandability. The method contains 2144 lines of code, while 1680 (78%) if it were annotated. The annotated code is introduced by 175 feature locations, thus 350 lines of the source code are variants of `#ifdef` directives. In total, 27 unique features were used and the nesting summed up to 78 nestings. The AB_{smell} is 138.7, while the *LocationSmell* with 137 is the highest sub-smell value. Besides the amount of nested annotated code and amount of feature locations, two patterns makes the method very hard to understand. The first pattern frequently appearing in the method, is shown in Listing 5.1. This is an example of the *Runtime if in combination* `#ifdef` pattern. Here, we can see a part of the method in which an if-else code fragment is annotated by four different features. Line 3705 introduces two features, which make the `if-else` block

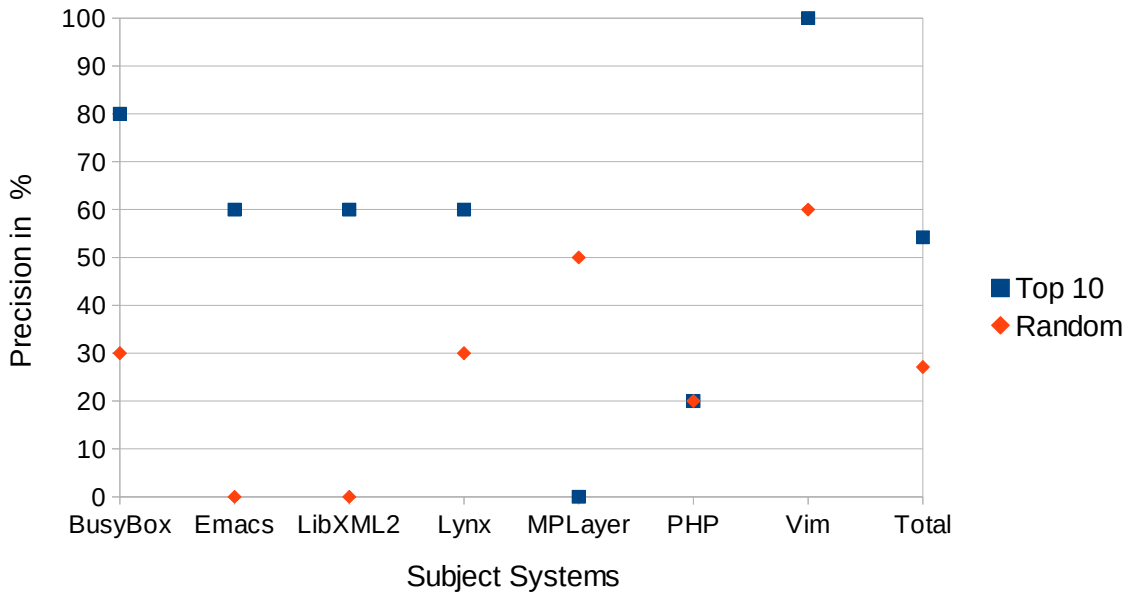

Figure 5.4: Annotation Bundle - Comparison of the detection precision between top 10 samples and random samples

optional. This single block is no problem for understanding this part of the method. However, line 3700 introduces a separate `if` block, which is another conditional for the previous block starting in line 6. Furthermore, line 3713 adds another `if-else` structure to the source code with the feature `FEAT_DIFF`, which changes the value of `n_extra`. All in all, the control flow of this not only changes with the configuration of the the four features, but again changes with the value of the parameters. In summary, the author argues that the control flow of this piece of code is very hard to follow.

The *Featurized God Function* is presented in Listing 5.2 as second pattern. Here, we can see the `capcol_lnum` introduced in line 2915. The variable is used in the beginning of the the method, similar to line 3080. Near the end of the file in line 5475, the variable is finally used again. At this point, a developer needs to have a complete overview of the method to understand if the variable was used or changed 2395 lines later. Depending on the configuration of all features, the number of possibilities to consider increases constantly.

The combination of both, the annotation of if-else statements and featurized variable assigning across the method, make such code hard to understand and therefore complex to maintain or change.

The `main` method of the file `xmllint.c` in LibXML2 is an example for a method with a rating of 0, that is, a method that partially exhibits smelly code. It contains 651 lines of code with 380 (58.4%) annotated lines. Across the method, 40 feature locations with 17 unique features are implemented. Five of those locations are nested

```
3700 .....
3701 #if defined(FEAT_LINEBREAK) || defined(FEAT_DIFF)
3702         if (draw_state == WL_SBR - 1 && n_extra == 0)
3703         {
3704       draw_state = WL_SBR;
3705 # ifdef FEAT_DIFF
3706       if (filler_todo > 0)
3707       {
3708           /* Draw "deleted" diff line(s). */
3709           if (char2cells(fill_diff) > 1)
3710         c_extra = _;
3711           else
3712         c_extra = fill_diff;
3713 #  ifdef FEAT_RIGHTLEFT
3714           if (wp->w_p_rl)
3715         n_extra = col + 1;
3716           else
3717 #  endif
3718         n_extra = W_WIDTH(wp) - col;
3719           char_attr = hl_attr(HLF_DED);
3720       }
3721 # endif
3722 .....
```

Listing 5.1: Annotation Bundle - Runtime if-else in combination with `#ifdef`. Excerpt taken from Vim screen.c

in other locations. The AF_{smell} adds up to 24.5, with 24.5 for the *LocationSmell*, 1 for the *ConstantsSmell* (meaning there are no complex locations) and 1.25 for the *NestingSmell*.

Listing 5.3 shows the implementation pattern, used for the majority of the method. This is an example of the *Repetitive Feature Code* pattern. Depending on the configuration of features, other command-line options are available for the subject system. Each location introduces new command-line options (line 3407, 3418 and ongoing). Each of these code pieces do not interact with each other. We believe that this pattern is easy for the developer to understand, maintain and extend.

However, certain parts have a negative impact. Similar to Listing 5.1, the method exhibits the combination of run-time `if-else` with `#ifdef` directives. Listing 5.4 shows an excerpt where not only the `if` s (line 3548 and 3582) are optional, but the conditional expression (3550 and 3584) can be extended by an additional feature. Therefore, the execution blocks (line 3552 and 3586) of the if-else statements are conditional on either two or three parameters.

In total, most of the methods code is easy to understand due to the repetitive design of the introduction of arguments. However, certain `if-else` statements are more complex because of the condition of the `if`-blocks.

The method `HTFileInit` in the file `HTInit.c` of Lynx is considered not smelly by the author. Out of 247 lines of code, 238 are annotated. These lines are implemented by 25 feature locations and 4 unique features. Out of these feature locations, 24 are nested.

```
2901 ...
2902 #ifdef FEAT_SPELL
2903 ...
2904     static linenr_T capcol_lnum = 0;   /* line number where "cap_col" used ↩
         */
2905 ...
2906 #endif
2907 ... // line 2918
```

```
3079 ....
3080    if (lnum != capcol_lnum)
3081        cap_col = -1;
3082    if (lnum == 1)
3083        cap_col = 0;
3084    capcol_lnum = 0;
3085 ....
```

```
5470 ...
5471 #ifdef FEAT_SPELL
5472     /* After an empty line check first word for capital. */
5473     if (*skipwhite(line) == NUL)
5474     {
5475           capcol_lnum = lnum + 1;
5476           cap_col = 0;
5477     }
5478 #endif
5479 ...
```

Listing 5.2: Annotation Bundle - Featurized God Function. Excerpt taken from Vim screen.c

```
3406 ...
3407 #ifdef LIBXML_READER_ENABLED
3408    else if ((!strcmp(argv[i], "-stream")) ||
3409             (!strcmp(argv[i], "--stream"))) {
3410         stream++;
3411    }
3412    else if ((!strcmp(argv[i], "-walker")) ||
3413             (!strcmp(argv[i], "--walker"))) {
3414         walker++;
3415              noout++;
3416    }
3417 #endif /* LIBXML_READER_ENABLED */
3418 #ifdef LIBXML_SAX1_ENABLED
3419    else if ((!strcmp(argv[i], "-sax1")) ||
3420             (!strcmp(argv[i], "--sax1"))) {
3421        sax1++;
3422        options |= XML_PARSE_SAX1;
3423    }
3424 #endif /* LIBXML_SAX1_ENABLED */
3425 ... // more features
```

Listing 5.3: Annotation Bundle - Repetitive Feature Codes. Excerpt taken from LibXML2 xmllint.c

```
3546 ...
3547 #ifdef LIBXML_SCHEMATRON_ENABLED
3548     if ((schematron != NULL) && (sax == 0)
3549 #ifdef LIBXML_READER_ENABLED
3550         && (stream == 0)
3551 #endif /* LIBXML_READER_ENABLED */
3552    ) {
3553 ...
```

```
3580 ...
3581 #ifdef LIBXML_SCHEMAS_ENABLED
3582     if ((relaxng != NULL) && (sax == 0)
3583 #ifdef LIBXML_READER_ENABLED
3584         && (stream == 0)
3585 #endif /* LIBXML_READER_ENABLED */
3586    ) {
3587    ....
```

Listing 5.4: Annotation Bundle - Run-time if-else in combination with `#ifdef`. Excerpt taken from LibXML2 xmllint.c

The AB_{Smell} severity value adds up to 26.4. The value is a result of 24.9 *LocationSmell* and 1.16 for both the *ConstantsSmell* and the *NestingSmell*.

While 96% of the method is annotated and therefore optional, all variable parts are implemented in two simple patterns. First, the main part of the method is nested in one surrounding feature. In Listing 5.5 in line 1058 , we can see the start of an `#ifdef` block that ends near the end of the file in line 1375. Inside this block, all other feature locations are nested. Therefore, if the feature `BUILTIN_SUFFIX_MAPS` is not set, most of the methods code is not active. Thus, the methods body is implemented using the *Optional Feature Stub* pattern.

Inside the surrounding feature, all other feature locations follow a simple and *Repetitive Feature Code* pattern. Similar to the code in Listing 5.3, each location introduces very similar code, such as the example in Listing 5.6. Each feature adds readable file suffixes to the subject system. Again, understanding or changing such code is very simple.

In conclusion, we found four frequently appearing patterns of introducing variability. On the one hand, *Featurized God Functions* (Listing 5.1) and *Run-time if with `#ifdef`* (Listing 5.2 and Listing 5.4) are bad patterns for implementing variants. The more code is introduced in such ways, the more complex the code gets. On the other hand, *Repetitive Feature Code* (Listing 5.3 and Listing 5.6) and *Optional Feature Stubs* (Listing 5.5) are patterns that are easy to read. The first simply introduces code in the same manner many times, the other decides whether or not the method does something or not.

With these patterns in mind, we can explain why Skunk was very precise on certain subject systems or not. The Top 10 samples of tool MPlayer proved to be not smelly in any kind, thus resulting a precision of 0%. Six out of ten samples were drawing methods that need some variations to work for different operating systems. All changes

```
1056 void HTFileInit(void)
1057 {
1058 #ifdef BUILTIN_SUFFIX_MAPS // line 1058
1059     if (LYUseBuiltinSuffixes) {
1060    CTrace((tfp, "HTFileInit: Loading default (HTInit) extension maps.\n"));
1061 ...
1062 #endif /* BUILTIN_SUFFIX_MAPS */ // line 1375
1063
1064     if (LYisAbsPath(global_extension_map)) {
1065    HTLoadExtensionsConfigFile(global_extension_map);
1066     }
1067
1068     if (IsOurFile(LYAbsOrHomePath(&personal_extension_map))
1069    && LYCanReadFile(personal_extension_map)) {
1070    HTLoadExtensionsConfigFile(personal_extension_map);
1071     }
1072 }
```

Listing 5.5: Annotation Bundle - Optional Feature Stub. Excerpt taken from Lynx HTInit.c

```
1102 ...
1103 #ifdef TRADITIONAL_SUFFIXES
1104    SET_SUFFIX1(".exe.Z",   "application/x-Comp. Executable", "binary");
1105    SET_SUFFIX1(".Z", "application/UNIX Compressed",   "binary");
1106    SET_SUFFIX1(".tar_Z",   "application/UNIX Compr. Tar",   "binary");
1107    SET_SUFFIX1(".tar.Z",   "application/UNIX Compr. Tar",   "binary");
1108 #else
1109    SET_SUFFIX5(".Z", "application/x-compress",  "binary", "UNIX Compressed"↩
       );
1110    SET_SUFFIX5(".Z", NULL,          "compress", "UNIX Compressed");
1111    SET_SUFFIX5(".exe.Z",   "application/octet-stream",   "compress", "↩
       Executable");
1112    SET_SUFFIX5(".tar_Z",   "application/x-tar",    "compress", "UNIX Compr.↩
        Tar");
1113    SET_SUFFIX5(".tar.Z",   "application/x-tar",    "compress", "UNIX Compr.↩
        Tar");
1114 #endif
1115 ...
1116 // more features
```

Listing 5.6: Annotation Bundle - Repetitive Feature Code. Excerpt taken from Lynx HTInit.c

were introduced in a repetitive manner similar to Listing 5.3. The remaining four samples had a similar structure.

None of the random 10 samples of both eMacs and LibXML2 were considered smelly. Again, we found out, that the methods either contained *Repetitive Feature Code* (9 for eMacs and 4 for LibXML2) or *Optional Feature Stubs* (1 for eMacs and 6 for LibXML2)

In contrast, the results for Vim showed a precision of 100%. The main problem with the code of Vim is the sheer amount of annotated code — 70% of the source code is implemented in 13413 feature locations. Nevertheless, when looking more closely on the code, we contend that the *Featurized God Function* is a frequently appearing pattern in the subject system. The median of lines of code of the top 10 samples is 1063.5, while the median of annotated code is 830 (80%) and the median of feature locations is 97. On average, this results in a feature location for every tenth line of code.

5.4.2 Annotation File

In the manual inspection process of the Annotation File samples set, the subject systems exhibited potential smelly files ranging from 29 files for Lynx up to 70 files for Vim. In total, 298 potentially smelly files were detected by Skunk. Again, we see a considerable difference in the number of detected smells. Thus, it shows how differently CPP annotations are used on the file level.

Name	#Pot	AF_{-1}	AF_0	AF_1	AF_{0+1}	% Precision
BusyBox	47	2 (4)	2 (6)	6 (0)	8 (6)	80 (60)
Emacs	30	5 (6)	2 (4)	3 (0)	5 (4)	50 (40)
Libxml2	48	1 (6)	6 (4)	3 (0)	9 (4)	90 (40)
Lynx	29	2 (9)	4 (1)	4 (0)	8 (1)	80 (10)
MPlayer	47	1 (7)	5 (3)	4 (0)	9 (3)	90 (30)
PHP	27	0 (7)	7 (2)	3 (1)	10 (3)	100 (30)
Vim	70	0 (3)	0 (2)	10 (5)	10 (7)	100 (70)
Total	298	11 (42)	26 (22)	33 (6)	59 (28)	84.3 (40)

Table 5.3: Results of the manual inspection for the Annotation File Smell, showing the number of potential smells, the ratings and the calculated precision. The first values indicate the top 10 samples, the values in parentheses indicate the results for the random samples.

Figure 5.5 shows the distribution of ratings across our subject systems. We notice that the number of actual code smells in the top 10 clearly tops the amount of code smells in the random 10 samples. This is clearly reflected in the comparison of precisions shown in Figure 5.6. The source code Vim and PHP exposed the largest number of actual code smells in the top 10, with each sample showing characteristics of the Annotation File smell. Skunk's precision for Emacs is the lowest, with half the samples being smelly files. In total, 59 out of 70 (84.3%) top 10 samples were considered as being

files that are negatively impacted with respect to understandability and readability. The lower rated random 10 samples of the subject systems exhibited a considerable amount of actual code smells, too. With one Annotation File, Lynx exhibits the least smelly files in the random-10 samples. With a 70% precision, Vim contains the the largest number of Annotation Files. The comparison the precision between the top-10 and the random-10 samples gives evidence, that the severity rating AF_{Smell} is a good indicator for the smelliness of files. All in all, the 84.3% precision for the top 10 samples is evidently higher than the 40% precision of the lower rated random 10 samples.

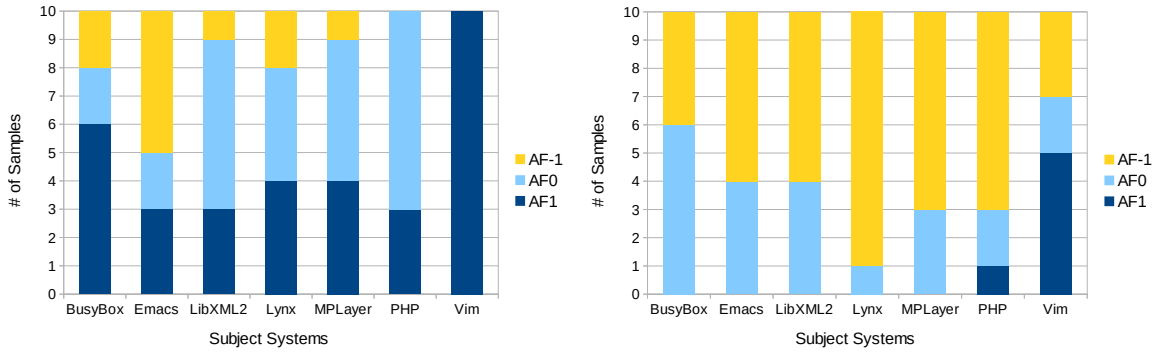

Figure 5.5: Graphical representation of the results of the manual inspection for the Annotation File Smell (Left - Top 10 samples, Right - Random 10 samples).

In the following, we take a look at an example file for each rating, and discuss the reason why they were smelly or not. The file `phpdgbparser.c` of PHP received a rating of 1, that is, a file where most of the source code is negatively impacted with regards to readability and understandability.. The file includes 1203 lines of code, with 697 lines being annotated ($\tilde{5}8\%$). These annotated lines are distributed across 127 feature locations with 60 unique features being used. With an accumulated nesting degree of 366, the file contains many deep nested feature locations. The severity rating AF_{Smell} is 78.7, with a *LocationSmell* of 73.6, a *ConstantsSmell* of 2.2 and a *NestingSmell* of 2.9.

A frequently appearing pattern in files with this rating is a huge amount of optional includes and macros. In Listing 5.7, we see such an example in the `phpdgbparser.c` file. Here, we see one of the `#ifdef` blocks that implements different macros and includes in an *Annotated Macro and Include Bundle* pattern. The main problem with this pattern is that it highly complicates the development or maintenance in the file. A developer working on this file, has to consider if the current configuration 1) enables the `#include` for the file he wants to use 2) enables the specific code fragment he is working on and 3) enables the exact functionality of functions of the included files. The nesting of this code block further complicates the development in this file. For example: the include of `stdlib.h` (line 333) is only active if `YYSTACK_USE_ALLOCA`, `YYSTACK_USE_ALLOCA`, `_STDC_` or `_C99_FUNC_` or `_cplusplus` or `_MSC_VER`

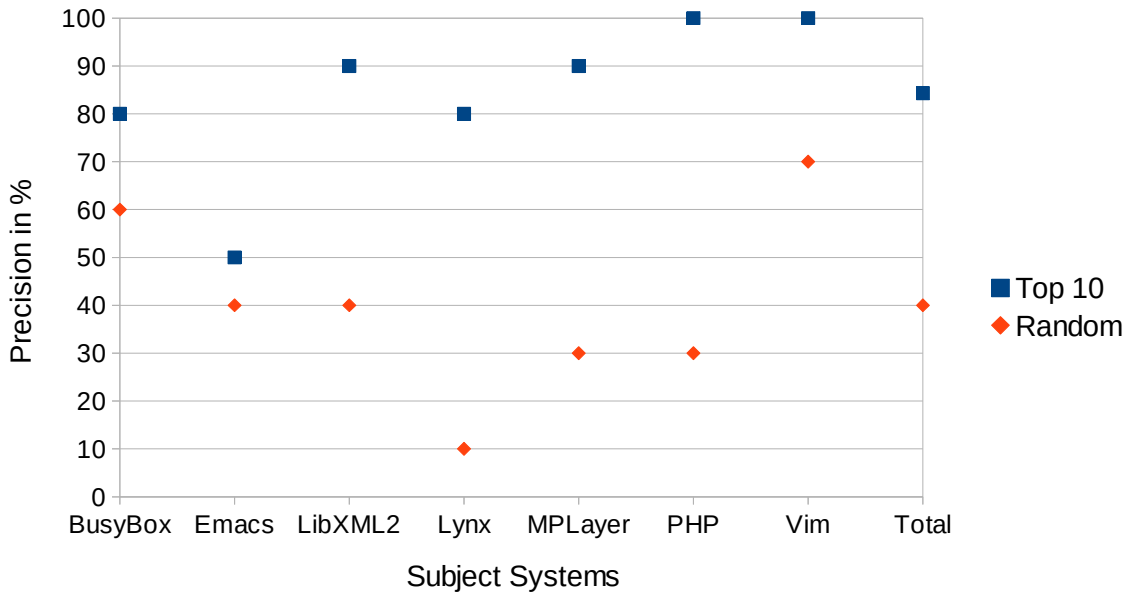

Figure 5.6: Annotation File - Comparison of the detection precision between top 10 samples and random samples

are active. Nevertheless, `_GNUC_`, `_BUILTIN_VA_ARG_INCR`, `_AIX`, `_MSC_VER`, `_ALLOCA_H` and `EXIT_SUCCESS` must be inactive in the configuration. This sums up to 12 different features, interacting with this `include`. Most of the files that received a rating of 1, contain multiple blocks similar to the example in Listing 5.7.

Another reason for the rating of `phpdgbparser.c` was the introduction of multiple function variants with *Function Header Variants* similar to Listing 5.8. Here, we can see the function header implemented in two ways (lines 774 and 778) depending on four features (line 772). This complicates the development in the same manner as the annotated `#include`. A developer always has to consider the current feature configuration, when using this function. Otherwise, it leads to unexpected results and errors. In combination with annotated `#include` s, it further complicates development when including this file in other files in the source system. Additionally, a change in the signature of the requires a consistent change in all occurrences of the method with regards to the feature.

In conclusion, the file `phpdgbparser.c` received a rating of 1 because of the massive use of *Annotated Macro and Include Bundles* in addition to many *Function Header Variants* throughout the file.

An example for a file with the rating of 0 is `pop.c` of Emacs. The file contains 1082 lines of code with 1078 (99.6%) being annotated. 16 unique features are used across 42 feature locations. The accumulated nesting degree is 65. While most of the functions consist of annotated lines of code, it was not considered to be completely smelly. The

```
317 // more featured includes and macros
318 ...
319 # ifdef YYSTACK_USE_ALLOCA
320 #  if YYSTACK_USE_ALLOCA
321 #   ifdef __GNUC__
322 #    define YYSTACK_ALLOC __builtin_alloca
323 #   elif defined __BUILTIN_VA_ARG_INCR
324 #    include <alloca.h> /* INFRINGES ON USER NAME SPACE */
325 #   elif defined _AIX
326 #    define YYSTACK_ALLOC __alloca
327 #   elif defined _MSC_VER
328 #    include <malloc.h> /* INFRINGES ON USER NAME SPACE */
329 #    define alloca _alloca
330 #   else
331 #    define YYSTACK_ALLOC alloca
332 #    if ! defined _ALLOCA_H && ! defined EXIT_SUCCESS && (defined __STDC__ ↩
    || defined __C99__FUNC__ \
333       || defined __cplusplus || defined _MSC_VER)
334 #     include <stdlib.h> /* INFRINGES ON USER NAME SPACE */
335       /* Use EXIT_SUCCESS as a witness for stdlib.h.  */
336 #     ifndef EXIT_SUCCESS
337 #      define EXIT_SUCCESS 0
338 #     endif
339 #    endif
340 #   endif
341 #  endif
342 # endif
343 ...
344 // more featured includes and macros
```

Listing 5.7: Annotation File - Annotated Macro and Include Bundle. Excerpt taken from PHP phpdgbparser.c

```
771 // more featured functions
772 ...
773 #if (defined __STDC__ || defined __C99__FUNC__ \
774      || defined __cplusplus || defined _MSC_VER)
775 static void
776 yy_symbol_print (FILE *yyoutput, int yytype, YYSTYPE const * const yyvaluep↩
    , void *tsrm_ls)
777 #else
778 static void
779 yy_symbol_print (yyoutput, yytype, yyvaluep, tsrm_ls)
780     FILE *yyoutput;
781     int yytype;
782     YYSTYPE const * const yyvaluep;
783     void *tsrm_ls;
784 #endif
785 {
786   if (yytype < YYNTOKENS)
787     YYFPRINTF (yyoutput, "token %s (", yytname[yytype]);
788   else
789     YYFPRINTF (yyoutput, "nterm %s (", yytname[yytype]);
790
791   yy_symbol_value_print (yyoutput, yytype, yyvaluep, tsrm_ls);
792   YYFPRINTF (yyoutput, ")");
793 }
794 ...
795 // more featured functions
```

Listing 5.8: Annotation File - Function Header Variants. Excerpt taken from PHP phpdgbparser.c

simple reason is, that the source code of this file is nested in one surrounding feature (line 24) as seen in Listing 5.9. Instead of only creating a *Feature Stub* in a method, the whole file becomes a *Feature File Stub*. If the feature `MAIL_USE_POP` is not active, the file contains no functionality. If the feature is active, all functions of this method can be used, since none of the features are enclosed in another feature location (line 159). Therefore, we do not consider the file to be very complex. Nevertheless, the file contains multiple *Annotated Include and Macro Bundles*, starting with line 28. This complicates the understandability of the file similar to the example in Listing 5.7.

Files with a rating of -1, were not considered to have a negative impact on understandability and readability because of at least one of two reasons. On the one hand, most of the sample files rated -1 contained an enclosing feature, thus creating a feature file stub similar to the feature `MAIL_US_POP` in Listing 5.9. Either the file contains functionality or not. On the other hand, Skunk considers feature locations in methods for files in its current iteration too. As a result all metrics of these feature locations are aggregated in to the severity rating. However, as per our definition, we only consider feature locations that introduce elements on the file-level (methods, global variables,...). Consequently, some of the potential samples of the Annotation File do not have any feature locations on the file-level. They are parents of Annotation Bundles, which we considered before. Therefore, we consider the specific methods as smelly, but not the file.

```
23 ...
24 #include <config.h> // line 24
25
26 #ifdef MAIL_USE_POP
27
28 #include <sys/types.h>
29 #include "ntlib.h"
30 // more includes and macros
31 ...
```

```
159 ...
160 popserver pop_open (char *host, char *username, char *password, int flags) ↩
        {...}
161 int pop_stat (popserver server, int *count, int *size) {...}
162 ... // more non-featured functions
```

```
1588 ...
1589 #endif /* MAIL_USE_POP */
1590 ...
```

Listing 5.9: Annotation File - Feature File Stub with Annotated Macro and Include Bundle. Excerpt taken from Emacs pop.c

In conclusion, we detected three frequently appearing patterns for the Annotation File across all subject systems. The *Annotated Macro and Include Bundle*, which implements optional macros and optionally includes files, and the *Function Header Variants*, a pattern that introduces different function headers (e.g. changing parameters) for many functions depending on the active feature configuration. Both patterns complicate maintenance and evolution of the source file, since the a slight change in the feature configuration may lead to unexpected results and errors. The *Feature File Stub* pattern is considered to have a negative impact. Depending on the configuration of the file, the file either contains the source code of the file, or is (mostly) completely empty Therefore, it as easy to read and understand as a non-variable source file.

These patterns are the main reason for our results in our manual inspection process in Table 5.3. Both, Vim and PHP, received a precision of 100% for the top 10 samples. All of Vim s files received a rating of 1, which is a result of the combination of massive use of *Annotated Macro and Include Bundle* and *Function Header Variants* in all samples. The amount of annotated code and feature locations, as seen in Table 5.1, further increases the complexity of all files. While six of the sample files of PHP in the top 10 samples received a rating of 0, the remaining four received a rating of 1. The six 0-rated samples contained mostly *Annotated Macro and Include Bundle*. The four 1-rated combined *Annotated Macro and Include Bundle* and *Function Header Variants*.

With a precision of only 10%, lynx received the least smelly samples for its random 10 samples. One file is the parent of Annotation Bundles. All of the remaining

samples contained and enclosing feature, and thus implemented the *Feature File Stub* pattern.

5.4.3 Large Feature

For the Large Feature, we need to find a threshold at which point a feature is considered to be large. For this purpose, we calculated the 75th percentile (3rd Quartile), 95th percentile and 97.5th percentile of the LOFC and NOFC of the 5017 features of all subject systems.

The results for the LOFC can be seen in Figure 5.7. The 75th percentile (Figure 5.7 (1)) resulted in 54 lines of feature code, that is, 75% of all features have less than 54 lines of code. We can see that 1307 features are bigger than this, which is still a huge number of features. Therefore, we increased the percentile calculation to 95 (Figure 5.7(2)), whereas the threshold increased to 486 lines of feature code. While the number of features larger than this decreased to 311. While this is a rather low percentage of features, we contend that the threshold calculated with the 97.5th percentile is an even better fit. First, the 97.5th percentile is the upper limit when calculating the 95% confidence interval. The calculation of this interval is a typically used method in applied practice [Z+99]. Second, between 400 and 1100 the number of features still decreases rather faster (e.g., -30 from 500 to 600 lines of code). From 1122 lines of code up to the maximum of 13596 lines of code, the number of features decreases very slowly (0-3 per 100 lines of code) in the used interval. This is a sign, that these features are getting more and more uncommonly large. Therefore, we are confident that this threshold can be used as an essential characteristic of a Large Feature. The results for the NOFC threshold are shown in Figure 5.8. Again, we can see that a large number of features contains more feature constants than the 75th percentile (Figure 5.8(1)). The threshold increased to 29 and 56 with the 95th and 97.5th percentile, respectively (see Figure 5.8(2) and (3)). Similar to the LOFC threshold, we contend that the 97.5th percentile threshold is the best fit for similar reason. In conclusion, the LOFC limit is $L_{NOFC} = 1122$, while the NOFC limit is $L_{NOFC} = 56$.

In Table 5.4, we present the results for the outlier test. The table contains the number of features per subject system as well as the number of features that either exceeded the LOFC or NOFC limit. The last two columns contain the number of features that exceeded both limits, and the highest rank a Large Feature reached, respectively.

We can see that all systems but BusyBox exhibit Large Features when using these thresholds. Vim and LibXML2 are the subject systems with the highest number of annotated code. Additionally, they also contain the most Large Features. BusyBox in contrast, is the only subject system in which no feature exceeded both thresholds at the same time. Only three features are *large* regarding their lines of feature code. Emacs is the system with the smallest number of Large Features. One feature exceeded both thresholds, but only one feature contains more than 1122 lines of code. In total, only 1.49 % of all features in our subject system are Large Features.

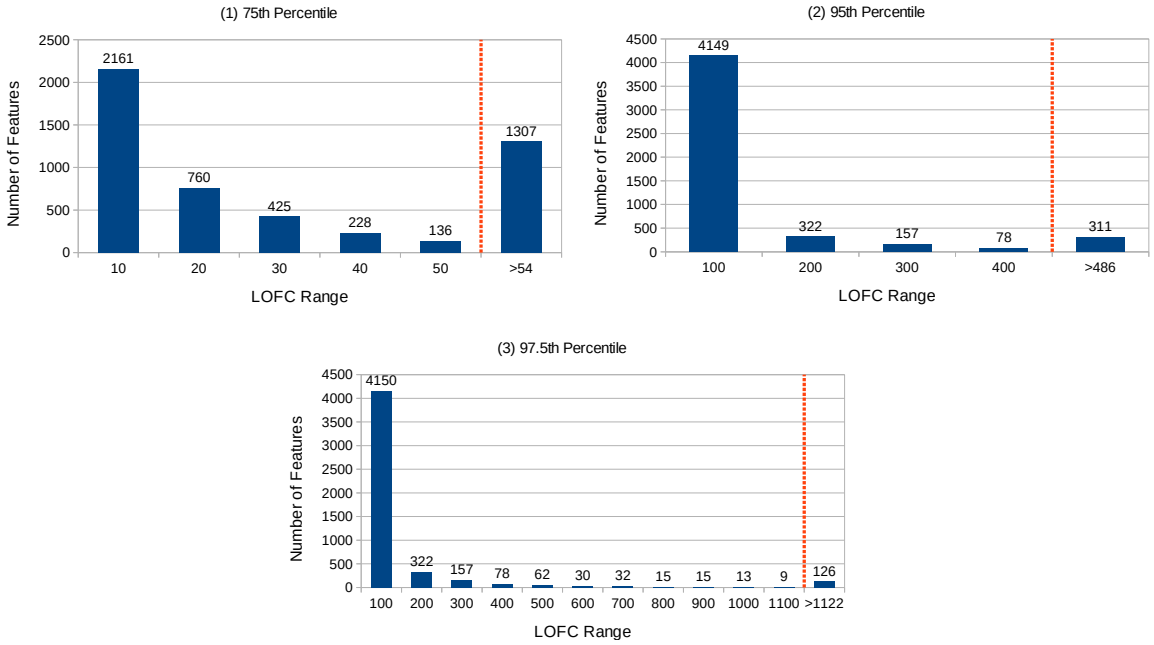

Figure 5.7: LOFC Outlier Test for all Features. The Red Line shows the (1) 75th Percentile, the (2) 95th Percentile and the (3) 97.5th Percentile

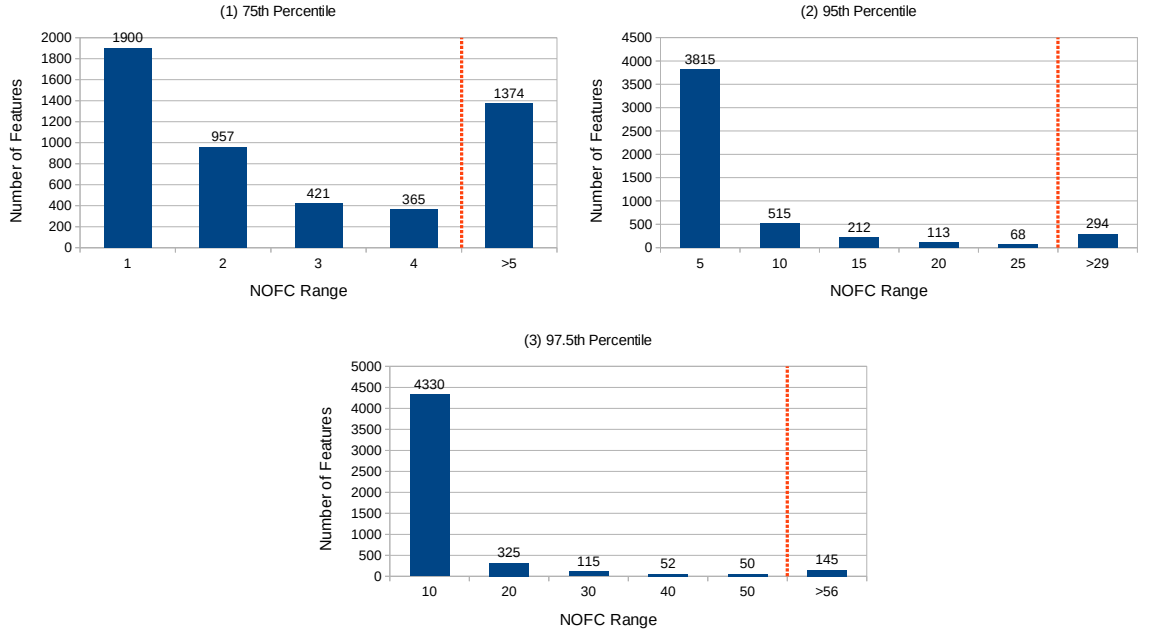

Figure 5.8: NOFC Outlier Test for all Features. The Red Line shows the (1) 75th Percentile, the (2) 95th Percentile and the (3) 97.5th Percentile

Name	# Feat	# Feat > L_{NOFC}	# Feat > L_{LOFC}	# Large Features	Highest Rank
Busybox	1263	3	2	0	-
Emacs	353	1	4	1	1
LibXML2	406	22	26	16	24
Lynx	519	10	12	6	6
MPlayer	1079	20	12	8	15
PHP	556	9	4	4	5
VIM	841	61	65	40	61
Total	5017	126	113	75	260

Table 5.4: Results for the Statistical Analysis of Large Features with $L_{NOFC} = 1122$ and $L_{NOFC} = 56$

We ordered all features by their severity rating LG_{smell}. As a consequence, features with higher a rating are higher ranked than features with a low rating. In our results, we examined that the lowest rank of a Large Feature is 260 with an LG_{smell} of 0.0075. Half of the Large Features (32) are ranked above rank 38. That means, the the density of features considered to be large is increasing the higher the LG_{smell} is. We can conclude that LG_{smell} severity rating can be used to rank potential features suffering from this smell.

5.5 Discussion

In the previous section, we performed a detection for the smells Annotation Bundle, Annotation File and Large Feature in seven subject systems. Our manual inspection proved that each subject system exhibited these variability-aware smell. Furthermore, we discovered frequently appearing high-level implementation patterns using `CPP` annotations. In his section, we we report more on qualitative observations that we made during our manual inspection of the Annotation Bundle/file samples and the statistical approach for the Large Feature. In detail, we relate our findings to our research questions proposed in Chapter 1. We discuss if Skunk is in fact able to detect variability-aware code smells and can order them after its severity. Furthermore, we show and define the recurring higher-level patterns we examined in the subject systems.

RQ1: Is a metrics-based detection algorithm able to find meaningful instances of variability-aware code smells?
Yes, the metrics-based approach of Skunk, proved to be able to detect meaningful instances. In total, 57 out of 140 (40.7%) manually inspected samples for the Annotation Bundle were considered as smelly, that is, a sample that has a negative effect on readability and changeability. The total precision for the top-10 rated samples was 54.2%, while the precision for the random-10 samples was 27.1% We examined the highest precision for Vim with 100% for the top-10 rated samples and 60% for the random-10 samples. PHP does not have any Annotation Bundles in the top-10 samples, because of the repetitive feature structure of all samples.

The manual inspection of potential Annotation Files showed that 87 out of 140 (62.1%) samples are in fact Annotation Files. Out of 70 top-10 rated samples 59 (84.3%) were actual Annotation Files. All of the top-10 samples for Vim and PHP were rated as smelly. We examined the lowest precision for the top-10 samples in Emacs with 50%. The total precision for the random-10 samples was 40%. With a precision of 70%, Vim again contained the most smells in the random-10 rated samples. Lynx contained the least actual smells in the random-10 samples with a precision of only 10%.

The differences in the amount of potential code smells for the Annotation Bundle and Annotation File show that there is a considerable heterogeneity in how to implement variants with cpp annotations.

With the thresholds (1112 LOFC and 56 NOFC) we calculated for the Large Feature we found 75 smells in the 5017 features of all subject systems. Vim contained the most Large Feature with 40, while BusyBox did not contain any Large Feature. The calculations have shown that 75% (3762) proved to be smaller than 55 lines of code and were not used in more than 5 feature locations. From this can conclude, that it is a common practice to implement rather small features. Large Features are more uncommon but they size prove to be a problem for comprehension.

RQ2: Is it possible to give instances found by a variability-aware code-smell detector a ranking, indicating their severity?
Yes, it is possible to calculate severity ratings which coincide with the human assessment of the proposed code smells. In our evaluation, we split the samples set into 10 samples consisting of the top-10 rated samples. The remaining 10 samples were randomly chosen from the remaining potentially smelly instances. For most subject systems, the precision of detecting actual code smells was higher for the top-10 rated samples. Only the manual inspection results of MPlayer for Annotation Bundles showed a lower precision for the top-10 samples (0% vs. 50%). For PHP, the precision of both samples sets was equal. For the remaining subject systems, the precision was at least 30% higher than the precision for the random-10 samples. In general, Skunk showed a 27.1% higher precision for the Annotation Bundle and a 44.4% higher precision for the Annotation File while comparing both samples sets. All actual Large Feature are above the 260th (5017 in total) position in the ranking, while 33 out 75 Large Features are above the 38th position. That means, that a higher rating for a Large Feature is potentially more smelly than a lower-rated feature.

RQ3: Does the smells exhibit recurring, higher-level patterns of usage?
Yes, during the manual inspection of the samples set, we examined recurring implementation patterns using Cpp-annotations. In the following we characterize these patterns, starting with patterns with a negative impact on readability and changeability. Afterward, we define patterns with a neutral or positive effect.

Negative Patterns

- ***Featurized God Function:*** A *Featurized God Function* is a function of massive length with a huge number of annotations and annotated source code. In many cases in such functions, parameters are initialized in annotations at the beginning of the function. Then, at the end of the function the parameter is assigned a value while in between the initialization and assignment much functionality is implemented. A developer has to have full knowledge of such a lengthy function and all its possible feature configurations to be sure a change of this parameter, for example, does not result in errors.

- ***Run-time `if` with `#ifdef`-Blocks:*** The *Run-time `if` with `#ifdef` blocks* pattern is characterized by `if` and `else` statements or conditional expressions that are introduced or changed by `#ifdef` annotations. The execution block of the `if` statement is then not only dependent on the conditional expression, but also the features or other introduced `if`s. This leads to multiple different control flows for this block of statements, which is hard to follow by developers depending on the amount of variations.

- ***Annotated Macro and Include Bundles:*** The *Annotated Macro and Include Bundles* pattern is characterized by a large number of `#include` and `#define` directives that are enclosed in `#ifdef` blocks, often nested in multiple `#ifdef` blocks. Depending on the feature configuration functions of other files and macros are either usable or not. This pattern presents a difficulty for developers. A developer working on a code fragment has to consider if the current configuration 1) enables the `#include`/`#define` for the file he wants to use 2) enables the specific code fragment he is working on and 3) enables the exact functionality of functions of the included files.

- ***Function Header Variants:*** The *Function Header Variants* pattern is characterized by the introduction of varying function signatures (or its parameters) using annotations. The signature of a function switches depending on the configuration. The pattern is used to change the number of function parameters or the types of parameters. The main problem with this pattern is that it must always be ensured that a call to the function is working for every feature configuration. A slight change or fix of the function requires developers to check each call site. Moreover, errors due a change of the signature can occur at many call sites in different code files. Finding and fixing the error can take a lot of development resources. As a consequence, using the *Function Header Variants* pattern can be very error-prone, since errors might occur in a specific configuration.

Neutral Patterns

- ***Repetitive Feature Code:*** *Repetitive Feature Code* is a code fragment, where multiple sequentially introduced code clones are annotated by a different feature. Since each annotated code block is very similar to other blocks in the function, it is very easy to understand. The pattern is mostly used for initialization functions

in which parameters or configuration values are set. Another use is to introduce command-line options that are only present in particular variants of the software system.

- ***Feature Stub:*** A *Feature Stub* is a file or function where one feature location encloses most of a function's/file's source code. Depending on the feature configuration the function or file is either empty and contains no functionality, or is fully available, similar to a non-annotated function/file. This pattern can be used, for example, to ensure that the logic of a function is not accidentally used in a wrong feature configuration. For a developer, this pattern is not problematic. On the one hand, the feature stub is either empty or fully functional. Therefore, it can be understood as a standard function/file or an empty one. On the other hand, it saves developers to accidentally use the function or file in wrong configuration thus avoiding errors.

5.6 Threats to Validity

While we designed the methodology of how to perform the evaluation with care, certain threats remain. To define a code smell, we had to define certain configuration parameters. These thresholds may lead to missing smells. However, we chose conservative parameters to lower the risk of missing actual smelly code fragments. Furthermore, the sampling of potential code smells may lead to an amount of samples that is not representative or is too small. We countered these risks by dividing the samples into peak values (top 10 samples) and a randomized samples set over the complete value domain (random 10 samples). Moreover, rating the samples by manual inspection is always biased on the experience and opinion of the author. As such, the samples were not inspected by another, independent person. In the scope of this thesis, it was not possible to find qualified personnel, who would invest the substantial amounts of time for manual inspection. Nevertheless, I made sure that each sample was carefully analyzed and that the reasons were clearly described. While, we made sure to explain why the thresholds for the Large Feature are a good fit, it can still be regarded as a bit subjective. In future research it is necessary to confirm these thresholds by qualified personnel.

The selection of the subject systems is not representative of all domains. However, since we chose subject systems of different domains, we made sure the that the results are not biased to a single domain. Therefore, the results can at least be generalized to similar domains.

5.7 Tuning the Parameters

With the previous step, we showed that our detection works, the proposed variability-aware code smells exist and how often they occur in our subject systems. Furthermore, we got the precision of our detection algorithm for each code smell. The defined parameters, however, were chosen conservatively to reduce the number of false-negatives. In

this step, we want to find more suitable parameters for the Annotation Bundle and Annotation File, which will not exclude any of the confirmed code smell instances, but at the same time, remove false-positives. For the Large Feature, we do not need to find more suitable values. We performed an outlier test to define at which thresholds a feature can be seen as large. To this end, we increase each configuration parameter for each smell independently until one of our validated code smells is no longer reported in at least on our our subject systems. Then, with the new combination of parameters, we perform the detection on all subject systems again and use the detection results for a new manual inspection.

The detection results of the detection with the optimized parameters won't contain each potentially smelly sample of the previous detection. The new thresholds will be slightly higher than before, thus removing potentially smelly samples from the results. To get the same amount of samples to be manually inspected, we include samples from the detection results with similar severity ratings at the position of the removed samples. We believe, that while not all newly chosen samples are actual code smells, some samples will show signs of smelly code. Consequently, this slightly increases the precision of Skunk.

Results

- Annotation Bundle:
  Based on the results of the manual inspection, the optimized configuration parameters are as follows: $LOAC_{ratio} = 52\%$, $NOFC_{dup} = 2$, $NOFL = 4$ and $ND = 1$. With the detection, we received 16 less potential smells while none of the top 10 samples were missing from the samples set. One out of four replacing samples received a rating of 0 in contrast to the -1-rated sample of the old set. Consequently, the precision for the random 10 samples increased by 1.5% (28.6%) in total.

- Annotation File:
  Based on the results of the manual inspection, the optimized configuration parameters are as follows: $LOAC_{ratio} = 50.73\%$, $NOFC_{dup} = 6$, $NOFL = 2$ and $ND = 1$. Again, we examined no change in the top 10 samples. All in all, the amount of potential code smells decreased from 297 to 263. Four false-positive samples (out of 17 samples) were replaced by 0-rated samples, thus increasing the precision for the random 10 samples to 45.71% in total from 40%.

As the results show, with optimizing the configuration parameters based on the previous results, we got a slight increase in precision. The reason for this is, that some low-rated false-positives were removed from the samples set. While not all replacements are differently rated, 5 five of the inspected samples show signs of smelliness.

Threats To Validity

While the new configuration parameters increased the precision for our selection of subject systems, we cannot generalize the results to other systems. For now, we can

only confirm the advantages of these parameter values for this selection, since it operates as a training set. In future research, the results of our parameter tuning experiment must be validated on a different set of subject systems.

6. Related Work

Code smells are a good indicator of when to use refactoring methods [Fow99]. We presented an automated approach to detect variability-aware code smells in software systems. In previous work Schulze et al., addressed variability-aware refactoring [STKS12] while Liebig et al. advanced in automated variabiliy-aware refactoring in the presence of CPP annotations [LJG+15]. Our work is complementary to theirs, in that it is necessary to find code smells to actively refactor them.

It is not always clear if the use of CPP annotations directly results in a negative effect on source code, and therefore many researches have investigated the potential effects. Spencer and Collyer, for example, argued against the use of `#ifdef` annotations, but focused on a single system[SC92]. In another research, Ernst et al. [EBN02] analyzed CPP patterns empirically but focused on macro extensions and techniques to replace preprocessor annotations. None of this research however, addressed specific patterns of preprocessor usage or the automatic detection of code flaws. Medeiros examined bugs that are related to the use of preprocessor annotations[MRG14][MKR+15]. In combination with other researches, he came to the conclusion that variability-related bugs are easier to introduce, harder to fix and more critical. This research underlies the importance of our work in that it is necessary to find complex source code patterns. We investigated sources of complexity in annotated source code. Other researchers have analyzed scattering, tangling, and nesting (as well as other properties) of `#ifdef` directives in highly configurable systems [LAL+10][HZS+15]. In contrast to this work, their analysis are statistical in nature, and do not discuss methods to detect concrete patterns of misuse.

A large body of research has been focused on metrics and tools to detect code smells in object-oriented software [VEM02][ML06][KVGS09][MGDLM10]. Additionally, Abilio et al. examined metrics for the detection of code smells in feature-oriented programming software systems[APFC15]. Furthermore, Figueiredo et al. centered on an approach to find modularity flaws based on metrics and heuristics [FSGL12]. In 2011, Schulze et

al. addressed the problem of the code smell Duplicated Code in combination with preprocessor variability [SJF11]. Similar to the research in this thesis, they focused on the automatic detection of code smells. However, other research was limited on either one smell [SJF11], cross-cutting concerns [FSGL12] or non-variable software systems [VEM02][ML06][KVGS09][MGDLM10]. In contrast, we proposed a general metrics-based approach and implemented it in as a detector. Furthermore, we focused on a commonly used variability mechanism, and validated our approach with an evaluation on seven real-world software systems.

7. Conclusion

Variability plays an important role in the development of 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 thesis, we proposed a metrics-based approach and its implementation to detect three variability-aware code smells in seven subject systems from different domain. Skunk, our variability-aware code smell detector, gathers metrics such as lines of annotated code or nesting degree of all features of a system that uses preprocessor annotations. A user can define code smells with thresholds and ratios based on the metrics Skunk gathers. The code smell configurations can then be used to detect code sections that exceed the parameters. After performing the detection, Skunk presents the results in a location file which includes precise information where to find smelly parts of a software system. In another file, our tool orders the potentially smelly code fragments by a rating indicating the severity of the negative impact on readability and changeability.

With Skunk we wanted to 1) validate our metrics-based approach and 2) detect three variability-aware code smells — the Annotation Bundle, the Annotation Bundle, and the Large Feature. To this end, we performed a comprehensive manual inspections for the Annotation Bundle and File. For the Large Feature, we used a statistical approach to decide at which point a feature is considered to be *large*. For the manual inspection we created a sample-set for each subject system based on the detection for both smells. The samples set consists of 10 samples of the top-10 rated potential code smells and 10 samples of randomly chosen samples of lower ratings. Then, each sample was manually assigned a rating. Our results show that Skunk is able to

find meaningful instances for both Annotation smells. The precision for Annotation Bundle ranged from 0% up to 100%, while the average precision for the top-10 samples was 54.2% and 27.1% for the random-10 samples. The drop in the precision is a sign that the severity ratings for this smell can be used as indicator. The results for the Annotation File present a precision ranging from 10% to 100% depending on the subject system. The average precision for the top-10 was 84.3% and 40% for the random-10, again proving the usefulness of the severity rating.

During our manual inspection, we detected five recurring patterns in the use of preprocessor annotations. One the one hand, the *Featurized God Function*, the Run-time `if` with `#ifdef` blocks and the *Annotated Macro and Include Bundles* pattern had a negative impact on readability and changeability because of their structure. They either obscured the control flow or increased the complexity of the source code. On the other hand, the *Repetitive Feature Code* and the *Feature Stub* patterns were often used but had no negative impact. Their structure is easy to read and does not obscure the code.

For the detection of the *Large Feature*, we needed to find thresholds, since it is not clear when a feature is considered to be *large*. The more lines of code the feature implements, the more functionality is implemented by the feature. The more functionality is implemented, the longer a developer needs to fully understand a feature. Therefore, we want to find thresholds at which point a feature is large. To this end, we calculated these thresholds using a statistical approach. The maximum thresholds define at which point a value is an outlier. An outlier is equal to a Large Feature. We calculated a fitting percentile of the lines of feature code and number of feature constants over all 5017 features of all subject systems. In both cases, the 97.5th percentile proved to be the best fit. By this, we received two border values above which only 2.5% of all features are. We considered features that exceeded the *lines of feature code* and *number of feature constants thresholds* as Large Features. Out of 5017 feature samples, 75 are considered to be actual feature suffering of the Large Feature smell. Half of the features are ranked above the rank 38, indicating a good severity value.

8. Future Work

While we proved the existence of three variability-aware code smells, the evaluation was only focused to 7 subject systems. In future research, the evaluation must be extended to a wide variety of subject systems, to prove the general applicability of Skunk's metrics-based approach. Furthermore, the manual inspection was performed by the author only. As such, the problem of biased ratings of the samples samples exist. A questionnaire is a possibility to counter this. This questionnaire must be given to a wider audience of qualified developers, who then give feedback if the detected smells are problematic or not. Additionally, Skunk offers more metrics and configuration parameters than we used for the definition of our variability-aware code smells. Consequently, other possible code smells can be defined. Later, their existence can be proven with the help of Skunk.

In Section 5.7, we optimized the parameters of our proposed code smells configurations to improve the precision of Skunk. In the context of this thesis, the subject systems can be seen as subject systems. Therefore, the optimized parameters need to be applied to other subject systems to investigate whether the precision increases in comparison to our conservative set of parameter values.

Another possible research direction is to examine the evolution of variability-aware code smells across multiple versions of the subject system. With this multiple questions can be answered: Does the number of code smells increase over time? Does the number of code smells increase proportional to the size of the subject system? Are there (refactoring) phases, in which the number of smells decrease?

Code that is suffering of `OOP`-code smells is often changed and fixed than others [OCS10]. A question that is to be answered is, if there is a similar connection between annotated source code and variability-aware code smells.

While Skunk proved to be a functioning tool for detecting variability-aware code smells, there is room for improvements. In its current state, the calculation of the severity

ratings for the three code smells are hard-coded in the source code of Skunk. While the weighing mechanism allows users to tune the impact of sub-smell values, it is still very limited. One weighing value is always connected with one configuration parameter. Users have no possibility to freely use all possible metrics to calculate custom severity ratings for code smells. Implementing customizable severity ratings, would allow users to not only find possible other code smells, but would also allow them to order other code smells by its severity than the proposed three.

The current location file shows the user were to find potential code smells within files with the exact starting line of code. However, this approach demands the user to find the source file, open it with a text tool or IDE, and locate the line of code. Implementing Skunk as a plug-in into an IDE (e.g Eclipse) allows highlighting of source code parts suffering of variability-aware code smells This enables a faster manual detection of possible code smells and signalizes warnings during development for developer.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. (cited on Page 1, 10, 11, and 15)

[AKGA11] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 181–190. IEEE, 2011. (cited on Page 1, 6, and 34)

[APFC15] Ramon Abilio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. Detecting code smells in software product lines–an exploratory study. In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, pages 433–438. IEEE, 2015. (cited on Page 61)

[Ast03] Dave Astels. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference, 2003. (cited on Page 8)

[CDM+11] Michael L Collard, Michael J Decker, Jonathan Maletic, et al. Lightweight transformation and fact extraction with the srcml toolkit. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 173–184. IEEE, 2011. (cited on Page vii, 23, 25, and 27)

[CKM+03] Michael L Collard, Huzefa H Kagdi, Jonathan Maletic, et al. An xml-based lightweight c++ fact extractor. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 134–143. IEEE, 2003. (cited on Page 19 and 23)

[CN02] Paul Clements and Linda Northrop. Software product lines: practices and patterns. 2002. (cited on Page 10)

[DBDV+06] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346–355, 2006. (cited on Page 6)

[EBN02] Michael D Ernst, Greg J Badros, and David Notkin. An empirical analysis of c preprocessor use. *Software Engineering, IEEE Transactions on*, 28(12):1146–1170, 2002. (cited on Page 11 and 61)

[EFB01] Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001. (cited on Page 13)

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999. (cited on Page 1, 5, 6, 7, and 61)

[FS15] Wolfram Fenske and Sandro Schulze. Code smells revisited: A variability perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, page 3. ACM, 2015. (cited on Page 1, 2, 15, and 16)

[FSGL12] Eduardo Figueiredo, Claudio SantAnna, Alessandro Garcia, and Carlos Lucena. Applying and evaluating concern-sensitive design heuristics. *Journal of Systems and Software*, 85(2):227–243, 2012. (cited on Page 61 and 62)

[FSMS15] Wolfram Fenske, Sandro Schulze, Daniel Meyer, and Gunter Saake. When code smells twice as much: Metric-based detection of variability-aware code smells. In *Source Code Analysis and Manipulation, 15th IEEE International Working Conference on*, 2015. (cited on Page 33, 36, 38, and 40)

[HZS+15] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-based variability in open-source and industrial software systems: an empirical study. *Empirical Software Engineering*, pages 1–34, 2015. (cited on Page 25 and 61)

[KA09] Christian Kästner and Sven Apel. Virtual separation of concerns-a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009. (cited on Page 11)

[KPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 75–84. IEEE, 2009. (cited on Page 1 and 6)

[KVGS09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 305–314. IEEE, 2009. (cited on Page 1, 9, 61, and 62)

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, C Kastner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 105–114. IEEE, 2010. (cited on Page 1, 25, 35, and 61)

[LJG+15] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proceedings of the International Conference on Software Engineering*, 2015. (cited on Page 61)

[MGDLM10] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010. (cited on Page 1, 9, 61, and 62)

[MHB10] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14. ACM, 2010. (cited on Page 1)

[MKR+15] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the c preprocessor: An interview study (artifact). *DARTS-Dagstuhl Artifacts Series*, 1, 2015. (cited on Page 11 and 61)

[ML06] Radu Marinescu and Michelle Lanza. Object-oriented metrics in practice, 2006. (cited on Page 61 and 62)

[MRG14] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating preprocessor-based syntax errors. *ACM SIGPLAN Notices*, 49(3):75–84, 2014. (cited on Page 11 and 61)

[OCS10] Steffen M Olbrich, Daniela S Cruze, and Dag IK Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010. (cited on Page 6 and 65)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005. (cited on Page 10)

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23, pages 199–208. ACM, 1988. (cited on Page 23)

[Pea14] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014. (cited on Page 9)

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97–Object-Oriented Programming*, pages 419–443. Springer, 1997. (cited on Page 13)

[SC92] Henry Spencer and Geoff Collyer. # ifdef considered harmful, or portability experience with c news. 1992. (cited on Page 11 and 61)

[SJF11] Sandro Schulze, Elmar Jürgens, and Janet Feigenspan. Analyzing the effect of preprocessor annotations on code clones. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 115–124. IEEE, 2011. (cited on Page 62)

[SSS14] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 2014. (cited on Page 5)

[STKS12] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. Variant-preserving refactoring in feature-oriented software product lines. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 73–81. ACM, 2012. (cited on Page 61)

[TCADP10] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010. (cited on Page 6)

[TSFB99] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *ACM Sigplan Notices*, volume 34, pages 47–56. ACM, 1999. (cited on Page 8)

[vD+02] A van Deursen et al. Refactoring test code, in extreme programming perspectives, 2002. (cited on Page 8)

[VEM02] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002. (cited on Page 1, 9, 61, and 62)

[VSSA14] Tassio Vale, Iuri Santos Souza, and Cláudio Sant Anna. Influencing factors on code smells and software maintainability: A cross-case study. 2014. (cited on Page 6)

[Yam13] Atsushi Yamashita. How good are code smells for evaluating software maintainability? results from a comparative case study. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 566–571. IEEE, 2013. (cited on Page 1 and 6)

[Z+99] Jerrold H Zar et al. *Biostatistical analysis*. Pearson Education India, 1999. (cited on Page 52)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 30. November, 2015