Code Smells Revisited: A Variability Perspective

Wolfram Fenske
Otto-von-Guericke-University Magdeburg
Magdeburg Germany
wfenske@ovgu.de

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

ABSTRACT

Highly-configurable software systems (also called *software product lines*) gain momentum in both, academia and industry. For instance, the Linux kernel comes with over 12 000 configuration options and thus, can be customized to run on nearly every kind of system. To a large degree, this configurability is achieved through variable code structures, for instance, using conditional compilation. Such source code variability adds a new dimension of complexity, thus giving rise to new possibilities for design flaws. Code smells are an established concept to describe design flaws or decay in source code. However, existing smells have no notion of variability and thus do not support flaws regarding variable code structures. In this paper, we propose an initial catalog of four *variability-aware code smells*. We discuss the appearance and negative effects of these smells and present code examples from real-world systems. To evaluate our catalog, we have conducted a survey amongst 15 researchers from the field of software product lines. The results confirm that our proposed smells (a) have been observed in existing product lines and (b) are considered to be problematic for common software development activities, such as program comprehension, maintenance, and evolution.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Languages

Keywords

Design Defects, Code Smells, Variability, Software Product Lines

1. INTRODUCTION

Code smells are an established concept to describe that a software system suffers from design flaws and code decay [12]. Usually, a code smell intrinsically indicates the need for restructuring the source code by means of *refactoring*. As a result, the code becomes easier to understand and thus easier to maintain and evolve.

VaMoS '15, January 21 - 23 2015, Hildesheim, Germany

http://dx.doi.org/10.1145/2701319.2701321

Over the last decade, a variety of work has addressed the detection [30, 43] and correction of code smells [29]. Moreover, the impact of code smells on different aspects of software development, such as evolution, maintenance, or program comprehension, has been studied [1, 21, 22, 44]. Complementarily, anti-patterns have been proposed to describe more profound shortcomings, for instance, shortcomings that arise from the occurrence of several code smells in concert [7]. Hence, a certain maturity has been reached and thus, code smells are a well-established concept for traditional (mostly object-oriented) software systems.

In the recent past, however, highly-configurable software systems (also known as *software product lines (SPLs))* gained much attention in both, academia and industry. Such systems usually encompass a vast number of related programs (also called a *program family*), which are based on a common platform [8]. The notion of *features* is used to communicate commonalities and variabilities in a program family, and thus, to distinguish between particular programs. In this context, a feature is an increment in functionality, visible to a stakeholder.

An advantage of the SPL approach is that a feature is implemented only once but can be reused in many different programs, based on a user-specified configuration. As a result, the SPL approach improves, for instance, flexibility, time-to-market, or the reliability of programs. For implementing SPLs, different variability mechanisms exist, which basically follow one of two ways: *Composition-based* mechanisms aim at modularizing all code (and non-code) artifacts that belong to a particular feature. By contrast, *annotation-based* mechanisms provide a virtual separation of features by annotating the respective code just-in-place [19]. In either case, variability is implemented explicitly and thus, is part of the code base. This, in turn, may not only increase the complexity of the source code and thus, impede comprehension and maintenance [6, 41]. More than that, it also limits the application of existing techniques, such as source code analyses, because current approaches do not address variability.

We argue that it is necessary to take variability into account as a first-class concept for code smells and their detection. Only then can we extend the well-established foundations of code smells to the domain of configurable software systems. This is of special importance, because such systems are defined with longevity, which inherently leads to code decay during evolution.

In this paper, we address the aforementioned problem by revisiting code smells in the light of variability. To this end, we inject the notion of variability into existing code smells, resulting in an initial catalog of *variability-aware code smells.* Particularly, we make the following contributions:

- In this paper, we propose four code smells that take variability into account. Basically, we take existing code smells as a foundation and lift them up to SPLs, resulting in variability-

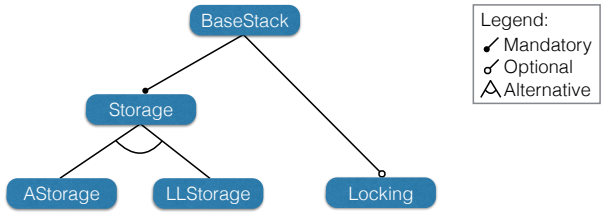

Figure 1: FODA feature model specifying valid configurations of a Stack product line

aware code smells.

- We discuss the occurrence and shape of these smells for two different variability mechanisms, the C preprocessor (CPP) and feature-oriented programming (FOP).
- We discuss possible (negative) effects of these smells on program comprehension, maintainability, and evolvability.
- To evaluate our code smells, we conducted a survey with 15 researchers that are experts on software product lines. Our results reveal that most of them (a) observed these smells in real-world systems and (b) acknowledge that such smells may hinder maintenance and evolution.

For all details on our survey as well as additional examples for variability-aware code smells, we provide a supplementary web page, available at https://www.isf.cs.tu-bs.de/cms/team/schulze/material/vacs/.

2. VARIABILITY MECHANISMS

In this section, we explain how variability in software product lines is managed on the domain level and how it is implemented on the code level. Moreover, we discuss the impact of variability on the code structure and, consequently, on code smells.

2.1 Variability Modeling

Features are one way of representing variability choices. Features denote increments in functionality, which can be common to all products, but can also be variable. This variability leads to (sometimes complex) relationships and dependencies among features. *Feature models (FMs)* are a common means to organize these relationships and dependencies in a tree structure [16]. In Figure 1, we show the FM of a simple stack product line (Stack SPL). Features in an FM can be *mandatory* (always present) or *optional*. Furthermore, they can form groups in order to express additional dependencies. For instance, the *Storage* feature has two child features: *AStorage* uses a dynamically resizable array, whereas *LLStorage* uses a linked list. Both features form an *alternative group*, that is, exactly one feature has to be selected for any variant. Finally, feature *Locking* constitutes an optional feature for concurrent access to the stack.

2.2 Variability Implementation

The variability modeled on the domain level has to be realized on the code level. To this end, different variability mechanisms exist, which can be subdivided into annotation-based and composition-based mechanisms. The difference between the two is the way of implementing features (a. k. a. feature code) in terms of modularization and separation of concerns [18]. As a result, the respective variability mechanism affects the way variable code is structured.

Annotation-based mechanisms.

Annotation-based mechanisms provide a *virtual separation* of concerns [18]. That is, all features of an SPL are implemented in a single code base. Annotations are used to mark code fragments that correspond to a given feature. In order to create a specific product, annotated code is removed or modified by means of a preprocessor [2]. As an example, consider the annotation-based implementation of the Stack SPL, given in Figure 2. The code is written in Java, and feature-related code is annotated with ANTENNA[1] preprocessor directives. ANTENNA directives are similar in syntax and semantics to C preprocessor (CPP) directives, with the difference that ANTENNA directives require an additional '//' prefix. For instance, the statements on Lines 9–18 implement `push()` and `pop()` for feature *Locking*, whereas Lines 20 and 21 contain the non-locking implementation. Both fragments are optional and their in-/exclusion is controlled by *configuration options*, which relate to features and are associated with preprocessor directives (a. k. a. `#ifdefs`). Consequently, the locking implementation is only included if macro `Locking` is defined. Conversely, if `Locking` is undefined, the code is removed during preprocessing.

```
1  class Stack<E> {
2  // #ifdef AStorage
3    List<E> store = new ArrayList<E>();
4  // #endif
5  // #ifdef LLStorage
6    List<E> store = new LinkedList<E>();
7  // #endif
8  // #ifdef Locking
9    void push(E e, Lock lock) {
10     lock.lock();
11     store.add(e);
12     lock.unlock();
13   }
14   E pop(Lock lock) {
15     lock.lock();
16     try { return store.remove(store.size()-1); }
17     finally { lock.unlock(); }
18   }
19 // #else /* !defined(Locking) */
20   void push(E e) { store.add(e); }
21   E pop() { return store.remove(store.size()-1); }
22 // #endif
23 }
```

Figure 2: Annotation-based implementation of the Stack SPL

Composition-based mechanisms.

Composition-based mechanisms *physically separate* concerns, which means that all artifacts (code and non-code) that belong to a certain feature are modularized into one cohesive unit [2, 18]. In *feature-oriented programming (FOP)*, this unit is called a *feature module*, and directly corresponds to a feature in the FM [5, 33]. There are several tools that support the feature-oriented implementation of SPLs, e. g., AHEAD [5], FEATUREHOUSE [3] and FEATUREIDE [42]. Beyond FOP, other composition-based mechanisms have been explored, for instance, component frameworks, plug-in architectures, and aspect-oriented programming [13, 40]. For our discussion, though, we focus on FOP due to its formal foundation, clear focus on physical separation of concerns, mature tool support, and availability of open-source case studies.[2]

In Figure 3, we show the feature modules of the FOP implementation of our Stack SPL. Module *BaseStack* introduces a class `Stack`, containing the methods `push()` and `pop()`. Furthermore, modules *AStorage* and *LLStorage* refine this class definition by adding field `store`. Finally, module *Locking* adds locking to methods `push()` and `pop()`. The locking code utilizes the FOP-specific keyword `original` to invoke the `push()` and `pop()` implementation provided by *BaseStack* (cf. Figure 3, Lines 4 and 9 in module *Locking*).

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

[2] See http://spl2go.cs.ovgu.de/ for a selection of SPL case studies.

Feature *BaseStack*

```
1  class Stack<E> {
2    void push(E e, Lock lock) { store.add(e); }
3    E pop(Lock lock) {
4      return store.remove(store.size()-1);
5  }}
```

Feature *AStorage*

```
1  class Stack<E> { List<E> store=new ArrayList<E>(); }
```

Feature *LLStorage*

```
1  class Stack<E> { List<E> store=new LinkedList<E>(); }
```

Feature *Locking*

```
1  class Stack<E> {
2    void push(E e, Lock lock) {
3      lock.lock();
4      original(e, lock);
5      lock.unlock();
6    }
7    E pop(Lock lock) {
8      l.lock();
9      try { return original(lock); }
10     finally { lock.unlock(); }
11 }}
```

Figure 3: Feature-oriented implementation of the Stack SPL

2.3 Effect of Variability on Code Smells

Upon closer inspection of the exemplary implementations of the Stack SPL, we observe several variability-related peculiarities, not unlike *code smells*. There are at least two flaws in the the annotation-based implementation (cf. Figure 2). First, Lines 11 and 16 of the *Locking* implementation are duplicated in Lines 20 and 21 of the non-locking code. Thus, whenever the non-locking code has to change, the locking code will likely have to change as well. The second flaw is that if *Locking* is enabled, both `push()` and `pop()` require an additional parameter, `lock`. Consequently, client applications cannot easily switch between locking and non-locking variants of the Stack SPL because that would require adapting every call site of `push()` and `pop()`. As others have previously observed, this is not desirable [34].

While the aforementioned peculiarities are absent from the composition-based example, another flaw occurs: Due to the fixed parameter list of `push()` and `pop()`, parameter `lock` is now unused in feature *BaseStack* (cf. Figure 3). However, `lock` cannot simply be removed, because it is required by module *Locking*.

As these examples show, design flaws may exist in the code structure of an SPL that are highly related to variability. Put differently, variable code structures create new opportunities to make bad design decisions. Considering the longevity of most SPLs, this increases the risk of code decay. This code decay, however, is not well captured by existing code smells, because these smells have been proposed for *single software products (SSPs),* that is, software systems with a fixed code structure. We argue that we have to make code smells aware of variability to capture design flaws in code with a variable structure. As a solution, in the next section, we propose *variability-aware code smells*.

3. A CATALOG OF VARIABILITY-AWARE CODE SMELLS

In this section, we first explain how we derived variability-aware code smells from SSP code smells. Then we present an initial catalog of four smells that were derived with this methodology.

3.1 Derivation Methodology

We have derived our proposed smells from well-known code smells described by Fowler et al. [12]. Specifically, we have considered how variability constructs, such as `#ifdefs`, can affect the language elements of the smell description, and how this will alter the code shape. For instance, the SSP code smell LONG METHOD describes a method with too many statements, indicating that the method is too complex to be understood easily. Applying our methodology to this smell, with a focus on annotation-based variability, our question was: "What will a LONG METHOD look like if some (or many) of the statements are guarded by `#ifdefs`?" This methodology works straightforward for many SSP smells besides LONG METHOD (e. g., DUPLICATED CODE, SWITCH STATEMENTS). However, it does not work for all smells. One counterexample is PRIMITIVE OBSESSION, which criticizes the use of primitive data types (e. g., char, int) when a class would be more appropriate. We have found no obvious way in which variability could affect this smell and a number of others.

In this paper, we focused on four SSP smells that have been found to occur regularly in source code. For those smells, we apply our methodology and distinguish between the two variability mechanisms CPP and FOP in case that their interactions with language elements matters. As we will show, there can be pronounced differences. We discuss them in more detail when we compare our variability-aware smells ANNOTATION BUNDLE and LONG REFINEMENT CHAIN.

3.2 Catalog

Next, we present four variability-aware code smells that we have derived using the method just described. For each smell, we start with a summary of the original SSP code smell, followed by a description of the derived smell. We further state which variability mechanisms (annotation-based, composition-based or both) the smell applies to. We then present an illustrative example and finally discuss potential problems for program comprehension, maintenance, and evolution that are caused by the respective smell. For our discussion, *maintenance* comprises bug fixes, quality improvements and other minor changes, whereas *evolution* means adding new functionality or making major modifications.

Inter-Feature Code Clones

Derived from: DUPLICATED CODE [12]

Code replication, also known as *code cloning*, is the practice of copying an existing piece of code and pasting it in another location with or without modification [35]. The result of code replication, DUPLICATED CODE, is not without problems. For instance, it has been identified as a source of subtle bugs [23] and has been linked to maintenance problems, such as the *inconsistent bug fix* (modifying some clone instances but missing others) [15].

Variability-aware description: There are two ways in which code duplication can occur in software product lines. First, code may be duplicated within a feature. However, the resulting clones do not depend on variability and hence are associated with the same problems as code clones in SSPs. More interesting is the second case, when there are two or more features that contain similar code. We call this smell INTER-FEATURE CODE CLONES. This smell can originate from intentional cloning (see Kapser and Godfrey for a description of common cloning strategies [17]). However, INTER-FEATURE CODE CLONES may also arise unintentionally. For instance, developers sometimes reimplement functionality that already exists in another feature because they are unaware of the existing solution.

Applies to: Annotation-based and composition-based mechanisms

Example: We already presented an annotation-based example of INTER-FEATURE CODE CLONES in the previous section (cf. Figure 2). The Graph Product Line (GPL)[3], a product line of classical

[3] http://spl2go.cs.ovgu.de/projects/49

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

Figure 4: Annotation Bundle in MySQL, file `mysys/thr_alarm.c`

graph algorithms [25], provides another example. The code excerpt can be found on our supplementary web page; for brevity, we omit it here. In the GPL, features *BFS* (breadth-first search) and *DFS* (depth-first search) contain an exact clone of the method `GraphSearch()`. This may appear surprising at first, as the two search algorithms are very different. Nevertheless, the GPL implementation is correct, as most of the work is performed by a helper method, which the features *BFS* and *DFS* implement differently.

Problems: We argue that INTER-FEATURE CODE CLONES are an even bigger obstacle than DUPLICATED CODE in SSPs. First, the aforementioned unawareness of clone instances in other features increases the likelihood of inconsistent changes. Secondly, the variability in an SPL adds another layer of complexity as the features containing the clones may be used in combination with many different sets of features. Consequently, when a bug is fixed or some other kind of modification is performed, the consistent propagation of the change to other features may not be necessary or even lead to semantic errors in other features. Hence, the developer has to verify for each feature in every valid configuration whether the change is both syntactically *and* semantically correct.

Annotation Bundle

Derived from: LONG METHOD [12]

The longer a method, the more difficult it is to understand [12]. A large number of statements can be used as a simple indicator of this smell [30].

Variability-aware description: An ANNOTATION BUNDLE is a method whose body consists of many variable parts. A large number of features controls which of these parts are included or excluded. On the code level, this results in many (groups of) statements that are annotated, e. g., using CPP directives. Several different annotations are involved, maybe even nested.

Applies to: Annotation-based mechanisms

Example: In Figure 4, we show a function implemented in C, which is heavily annotated with CPP directives. It is taken from the open source database management system MySQL, version 5.6.17.[4]

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

```
class Main { // Feature 'dmain'
  public static void process(Model root) throws /*...*/ {
    // layers extend this method for AST processing
  }
}
class Main { // Feature 'fillgs'
  public static void process(Model root) throws /*...*/ {
    original(m);
    ...
  }
}
class Main { // Feature 'propgs'
  public static void process(Model root) throws /*...*/ {
    original(m);
    ...
  }
}
class Main { // Feature 'formgs'
  public static void process(Model root) throws /*...*/ {
    original(m);
    ...
  }
}
class Main { // Feature 'clauselist'
  public static void process(Model root) throws /*...*/ {
    original(m);
    ...
  }
}
class Main { // Feature 'modelopts'
  public static void process(Model root) throws /*...*/ {
    original(m);
    if (modelMode) {
      try { harvestInfo(); }
      catch (IOException e) {
        JOptionPane.showMessageDialog(null, "Model" +
        " Harvesting Error -- see command line for" +
        " details", "Error!", JOptionPane.ERROR_MESSAGE);
        System.err.println(e.getMessage());
  }}}}
```

Figure 5: Long Refinement Chain in GUIDSL

Although the function is not very long in terms of C statements, there are five different CPP macros that control which of these statements make up a concrete implementation. Moreover, these macros are nested and sometimes negated. For instance, the statement on Line 23 is controlled by three different macros, two of which must be undefined for the statement to be included. Altogether, only few lines (e. g., the `if` condition on Lines 4–5) of the whole function are stable and thus, pervasive in each compiled program.

Problems: We argue that an ANNOTATION BUNDLE is difficult to understand for a certain configuration or in its entirety. Having many variable parts in the method body obscures the view on the core functionality. Moreover, each annotation requires additional knowledge of the macros that are involved. Hence, to comprehend an ANNOTATION BUNDLE, a developer has to work with many different abstractions on both the programming language level and the variability level.

Maintenance and evolution tasks are also hampered by heavily annotated methods. For instance, locating a bug is difficult if the exact configuration that exhibits the defect is not known. Moreover, developers have to take special care when changing heavily annotated code. Otherwise they might break the presence conditions of existing statements or introduce dangling references due to particular configurations they failed to consider.

Long Refinement Chain

Derived from: LONG METHOD [12]

Variability-aware description: The smell LONG REFINEMENT CHAIN is the composition-based counterpart of ANNOTATION BUNDLE. As such, it denotes a method with many variable parts due to feature refinement.

Applies to: Composition-based mechanisms

Example: In Figure 5, we show a LONG REFINEMENT CHAIN from GUIDSL [4]. GUIDSL is a product line configuration tool implemented in FOP. In our code example, we show the `process()` method of class `Main` and all of its refinements. The method is introduced as an empty stub by feature *dmain* and subsequently refined by five other features (*fillgs, propgs, formgs, clauselist, modelopts*). By contrast, the average refinement depth in GUIDSL is lower than one, that is, most methods are never refined. Each refinement of `process()` contains between three to nine additional lines of code and thus, contributes considerably to the overall method. Moreover, most of these refinements can occur in different combinations, depending on the feature selection.

Problems: In contrast to an ANNOTATION BUNDLE, with a

Feature *WeightedOnlyVertices*

```
1 public class Graph {
2   public void addAnEdge(Vertex start, Vertex end, int weight) {
3     addEdge(start, end, weight);
4   }
5   public void addEdge(Vertex start,  Vertex end, int weight) {
6     addEdge(start, end);
7     start.addWeight(weight);
8     /* More source code ... */
9 }}
```

Feature *DirectedOnlyVertices*

```
1 public class Graph {
2   public void addAnEdge(Vertex start, Vertex end, int weight) {
3     addEdge(start, end);
4   }
5   public EdgeIfc addEdge(Vertex start, Vertex end) { /* ... */ }
6 }
```

Figure 6: Latently Unused Parameter `weight` in the GPL

LONG REFINEMENT CHAIN, feature-specific parts of a method are encapsulated in refinements. Even though encapsulation is a favorable property, excessive refinement is problematic. First, as with ANNOTATION BUNDLEs, it is hard to understand the effect of a particular refinement for a concrete configuration. Secondly, when modifying an often-refined method or adding a new refinement, the developer must be aware of all existing refinements (and their combinations) and possible side effects of changing or adding code. Hence, we contend that methods with a LONG REFINEMENT CHAIN are harder to understand, maintain and evolve than methods with few refinements.

Interestingly, even though we have derived both, ANNOTATION BUNDLE and LONG REFINEMENT CHAIN, from the same SSP smell (LONG METHOD), they manifest themselves very differently on the code level. This illustrates how much the chosen variability mechanism affects the shape of an SSP code smell when it is transferred to an SPL context.

Latently Unused Parameter

Derived from: LONG PARAMETER LIST & SPECULATIVE GENERALITY [12]

A method with many parameters hampers program comprehension as each parameter increases the cognitive burden of the caller. Furthermore, a LONG PARAMETER LIST impedes evolution as it is frequently changed when additional data is needed [12].

SPECULATIVE GENERALITY, in turn, describes a situation in which there is functionality within a software system that is never used. Such functionality increases the complexity of the code but has no immediate benefit [12]. SPECULATIVE GENERALITY can take several forms, *unused parameter* is one of them.

Variability-aware description: In an SPL context, parameter lists are also subject to variability. Sometimes, a parameter of a method is optional, i. e., it is only needed by a certain feature, but is unnecessary for others. One way to deal with an optional parameter would be variable method signatures, that is, the parameter is only present if the corresponding feature is selected. However, as we have discussed in Section 2.3 and others have stated before [34], variable method signatures are problematic. Another solution is to forward-declare the optional parameter upon introduction of the method [34], even if it is used much further down in the refinement chain. The disadvantage of forward-declaration, however, is that for all features higher up in the refinement chain, the forward-declared parameter will be unused. This, in turn, is a form of SPECULATIVE GENERALITY.

Applies to: Annotation-based and composition-based mechanisms

Example: We have already presented a LATENTLY UNUSED PARAMETER in Figure 3 of the previous section. In this example, parameter `lock` was required by feature *Locking*, but unused in *BaseStack*. We show another example in Figure 6, taken from the GPL. Here, feature *WeightedOnlyVertices* extends class `Graph` with method `addAnEdge()` (see Lines 2–4). The third parameter of this method is an integer, `weight`. This is reasonable, as the feature provides support for weighted graphs. Indeed, `weight` is used by helper method `addEdge()` (cf. Lines 3 and 5–9). Feature *DirectedOnlyVertices* also provides a method `addAnEdge()`. In order to be compatible with feature *WeightedOnlyVertices*, the same signature is used. However, in the latter case, the parameter `weight` is confusing and, in fact, unused.

Problems: The natural assumption is that a parameter has some effect on the method's outcome. *Unused parameters* make a method harder to understand because they foil that assumption [12]. LATENTLY UNUSED PARAMETERs only foil that assumption in particular cases, which is at least as bad. Moreover, a LATENTLY UNUSED PARAMETER introduces coupling between callers of the method and the feature that requires the parameter. For instance, a client application of the GPL, which only uses feature *DirectedOnlyVertices* (and never *WeightedOnlyVertices*), will nonetheless have to supply a `weight` when calling `addAnEdge()`. This is not only confusing to developers of the client application. In addition, in order to understand the reason behind the extra parameter, the developers are forced to inspect *WeightedOnlyVertices* – a feature that is otherwise entirely irrelevant to them.

4. EVALUATION

While the descriptions of the variability smells in the previous section have been elaborated with care and are based on existing, well-established code smells, they may be biased to the author's point of view. Hence, before detecting such smells or proving their severity by means of empirical studies, it is necessary to verify whether the proposed smells "really are smells." To this end, we conducted a survey amongst experts in the field of software product lines. In the following, we provide information about the setup, the particular questions and the result of this survey.

4.1 Objectives

The objective of our survey was to receive feedback on our proposed variability-aware code smells from product-line experts. Particularly, we designed our survey with two questions in mind:

Q1: *Do our proposed smells exist in the design and implementation of SPLs?:* We assume that most participants of the survey have to deal with SPLs on implementation level, whether it be through teaching, analysis, implementation, or tool support. With this question, we aim to elicit which of the proposed smells the participants encountered in the described (or similar) form.

Q2: *Are our smells problematic with respect to different aspects of SPL development?:* Code smells do not just indicate a decay of source code, but also lead to problems regarding program comprehension, maintenance and evolution. Hence, we are interested in how the participants estimate the severity of our proposed smells with respect to these aspects.

4.2 Setup

In the following, we provide information about participants and concrete questions of our survey. The complete survey with all questions and corresponding answers is available at the supplementary web page (cf. Section 1 for the URL).

4.2.1 Participants

As target audience for our survey, we decided to send out our survey to participants of the international meeting on feature oriented software development (`www.fosd.de/meeting2014`), held at Schloß Dagstuhl in May 2014. The audience consists of PhD students, post-docs as well as professors, all of them working in the

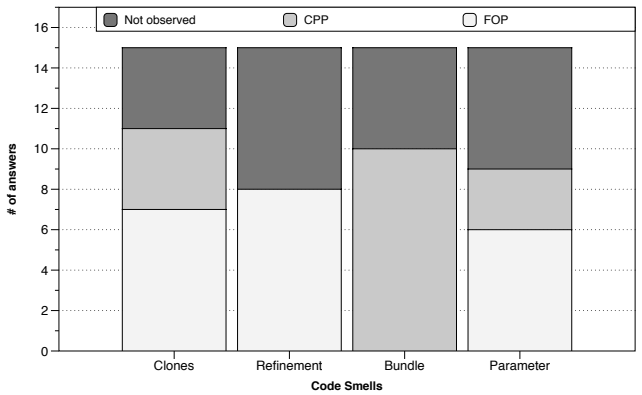

Figure 7: Results for the occurrence of variability-aware code smells wrt all answers (at most 15).

field of software product lines. We sent out our survey two weeks before the meeting and received 17 response sets. Two response sets were incomplete and thus, have been discarded. Moreover, to ensure that participants are eligible for our survey, we measured their programming experience for both, general purpose programming as well as SPL programming, based on the work of Siegmund et al. [11]. Particularly, we asked questions about programming experience, different programming paradigms, programming activity, and experience with different variability mechanisms. Based on the responses, we confirm that all participants are advanced programmers with a solid background in SPL implementation. Particularly, most of the participants are (very) experienced with feature-oriented programming and using the CPP, respectively. Thus, we argue that the responses we got are sufficiently reliable.

4.2.2 *Survey – Structure & Questions*

We divided our survey into four sections. Initially, we asked for *personal data* such as gender, age, or the current affiliation of the participant. Next, we had a larger section regarding the *background & experience* of the participant, which mainly focused on the above mentioned programming experience. In the third section, we asked questions about the *knowledge of code smells.* Given a basic knowledge of the concept, or even knowledge of concrete smells, it is probably easier to identify variability-aware code smells.

Finally, the last section contained questions about our proposed code smells and thus, constitutes the main part of our survey. For each of the variability-aware code smells in Section 3, we asked whether participants observed these smells, for which variability mechanism (if not specified by the smell), and in which kind of project (e. g., open source, industrial). Moreover, participants were requested to estimate the severity of each code smell for the following aspects: program comprehension, maintainability, and evolvability. Together with the survey, we handed out a short description of the code smells, similar to the one in Section 3. This description is available on the supplementary page.

4.3 Results

Next, we present the main results of our survey. For brevity, we only present results about variability-aware code smells, because this has been the main objective of the survey. Additional answers and comments are available at the supplementary page.

In Figure 7, we present the results related to the existence of our proposed smells. Basically, these results confirm that most of the smells have been observed "in the wild" by the participants. Particularly, 63% of the participants (average over all code smells) have observed at least one of the proposed smells. Concerning the variability mechanism, responses indicate that the smells occur equally in SPLs implemented with FOP and the CPP, respectively. Moreover, individual participants observed certain smells for proprietary mechanisms.

Beyond these observations, we asked for the impact of our proposed smells on different aspects of SPL development. Our results, which we show in Figure 8, reveal that most of the code smells are considered problematic. First, for program comprehension (cf. Figure 8 (a)), especially REFINEMENT CHAIN, ANNOTATION BUNDLE, and LATENTLY UNUSED PARAMETER have been judged to be problematic, while INTER-FEATURE CODE CLONES are seen as less of an issue. Secondly, the participants estimate that all code smells are mostly problematic for maintainability of SPLs (cf. Figure 8 (b)). Particularly, 61% on average estimate the severity of the smells "rather problematic" or even "very problematic." Similarly, participants judge these smells to impede evolvability of SPLs (cf. Figure 8 (c)).

4.4 Discussion

Next, we interpret and discuss the aforementioned results by relating them to the questions in Section 4.1.

Q1:*Do our proposed smells exist in SPLs?:* Based on the results, we argue that our proposed variability-aware code smells occur regularly in SPLs. For all smells, a considerable number of participants (between 50% and 70%) acknowledged that they faced the respective smells in SPL implementations. Moreover, participants observed these smells not only in toy examples; rather, they confirmed these smells also for open source and industrial projects, which is also reflected by the following two comments:

> [INTER-FEATURE CODE CLONES] *"Our industry partner is struggling with inter-feature code clones due to a lack of awareness. . . . "*

> [ANNOTATION BUNDLE] *". . . in Linux, I have observed that in some cases a lot of #ifdefs are used in a method and some of them are nested making the method longer and more complicated."*

Additionally, the fact that half of the affected projects were using the CPP gives rise to the conclusion that the proposed smells occur frequently *in the wild*.

Q2: *Are our smells problematic with respect to different aspects of SPL development?* Although our results reveal differences between particular code smells and the different aspects, the overall opinion is that our proposed smells impede program comprehension, maintenance, and evolution of SPLs. Particularly, up to 80% participants acknowledged possible problems. Of course, this result only reflects the personal opinion and experience, which may be somewhat subjective (e. g., no independent measurements exist). However, each participant has long-time experience with SPLs, and has worked on projects from different domains.

Overall, the results of our survey confirm that (a) the notion of variability is beneficial for reasoning about code smells and (b) that variability-aware code smells, as proposed by us, occur regularly and may impede SPL development.

5. RELATED WORK

Apart from us, Apel et al. introduced the term "variability smell" and present a summary of fourteen possible smells that may occur in different phases of SPL engineering, such as feature modeling, product configuration or SPL implementation [2]. The only overlap between their smells and ours is DUPLICATE CODE IN ALTERNATIVE FEATURES, a form of our INTER-FEATURE CODE CLONES that is restricted to alternative features. The list by Apel et al. is

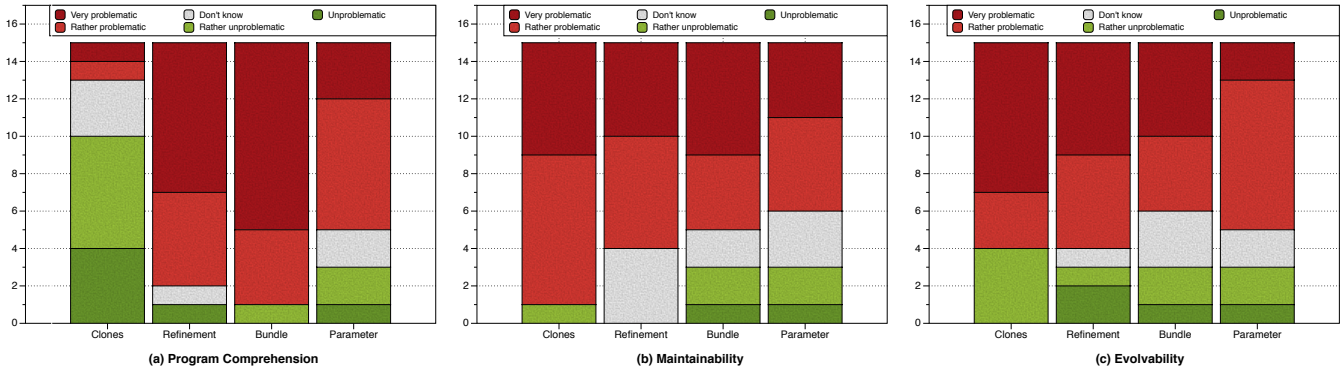

Figure 8: Results for how participants estimate the impact of our proposed code smells, regarding (a) program comprehension, (b) maintainability, and (c) evolvability.

an overview of problems that should be investigated in future research, rather than a fully evaluated and systematic methodology. Compared to their smells, we focus on code smells exclusively and provide a systematic methodology for defining such smells.

Beyond that, code smell definition and detection for product lines has been subject to recent work. Highly related to our approach is the work by Patzke et al., which deals with root causes of anomalies during SPL evolution [32]. They present a template of different high-level problems that may occur during SPL evolution and provide root causes for each problem. These root causes are given informally and may be combined to define code smells, similar to those we presented. Hence, we see synergy potential in our approach and theirs. However, our work differs in that we define concrete smells on the code level for different variability mechanisms, while they focus on higher level problems (which may manifest in source code) and on ways to mitigate them.

Next, different approaches exist that use SPLs as subjects for detecting code smells, as well as architectural decay [14, 26, 27]. Particularly, Andrade et al. identified traditional architectural smells in an SPL by means of a case study [9]. However, all of these approaches mostly focus on language-specific smells for OOP and AOP and put the focus on crosscutting concerns while neglecting variability. Moreover, they focus on detection rather than on providing a systematic methodology to define variability-aware code smells, as we do.

In a broader sense, our approach is related to foundational work on refactoring and anti-patterns/smells [20, 28, 31, 39]. However, these smells either lack the notion of variability in code [20, 28] or are specifically tailored to a concrete variability mechanism [31, 39]. In contrast, our smells explicitly take variability into account. Moreover, we discuss them for different variability mechanisms.

Beyond concrete code smells, a large body of work exists on observed shortcomings of CPP directives. Among other aspects, the negative impact of such directives on maintainability, program comprehension, and error-proneness has been discussed [10, 38, 41]. With respect to variability, Liebig et al. present empirical results on the use of `#ifdef` directives, specifically with a focus on granularity, scattering, tangling, and nesting of such directives [24]. While all this work is related due to the focus on shortcomings of the CPP, the proposed metrics, such as scattering, by themselves are too low-level to be considered code smells as code smells describe concrete patterns of misuse. However, the proposed metrics may be useful for future detection approaches, and may apply to other variability mechanisms besides the CPP.

Finally, in recent work, we provide details on the concrete smell DUPLICATED CODE in both, composition-based and annotation-based SPLs, and how to manage them [36, 37]. This work complements the smells proposed in this paper, because it gives empirical evidence for the existence of one of these smells.

6. CONCLUSION

Software product lines and other highly configurable software systems owe much of their configurability to variable code structures. This implementation-level variability introduces a new dimension of complexity, which in turn may lead to new kinds of design flaws. Code smells are established indicators of design flaws in source code. Such flaws make the code harder to understand, maintain and evolve. If not addressed, they may cause a decrease in programmer productivity or the introduction of actual faults. Unfortunately, existing smells have no notion of variability and thus, are insufficient to describe variability-related design flaws in highly configurable systems.

To address variability-related code decay, we propose an initial catalog of four *variability-aware code smells*. Our smells are based on established code smells but explicitly take variability into account. We consider both annotation-based and composition-based variability mechanisms, which makes our smells applicable to a large number of configurable systems. Moreover, we evaluated our smells by means of a survey, which was sent out to researchers from the field of software product lines. The majority of the participants confirm that our smells (a) have been observed in real-world systems and (b) constitute problems for understanding, maintenance and evolution.

Currently, we are working on an automatic detection framework, which will allow us to determine how frequently the proposed and other smells occur in existing systems. In future work, we will consider the impact of variability on further established code smells. We also plan to widen the scope of our research to anti-patterns, architectural smells and other indicators of faulty software design. Furthermore, we intend to investigate variability-aware refactorings to correct these design flaws. Eventually, this effort may result in a guideline to reduce complexity in highly configurable software.

7. REFERENCES

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *CSMR*. IEEE, 2011, pp. 181–190.

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

[3] S. Apel, C. Kästner, and C. Lengauer, "Language-independent and automated software composition: The FeatureHouse experience," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 63–79, 2013.

[4] D. Batory, "Feature models, grammars, and propositional formulas," in *SPLC*. Springer, 2005, pp. 7–20.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[6] I. D. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *WCRE*. IEEE, 2001, pp. 281–290.

[7] W. H. Brown, R. C. Malveau, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley & Sons, 1998.

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

[9] H. S. de Andrade, E. Almeida, and I. Crnkovic, "Architectural bad smells in software product lines: An exploratory study," in *WICSA Companion Volume*. ACM, 2014, pp. 12:1–12:6.

[10] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *IEEE Trans. Softw. Eng.*, vol. 28, no. 12, pp. 1146–1170, 2002.

[11] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring programming experience," in *ICPC*. IEEE, 2012, pp. 73–82.

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

[13] M. L. Griss, "Implementing product-line features by composing aspects," in *SPLC*. Kluwer Academic Publishers, 2000, pp. 271–288.

[14] E. Guimaraes, A. Garcia, E. Figueiredo, and Y. Cai, "Prioritizing software anomalies with software metrics and architecture blueprints," in *MiSE*. IEEE, 2013, pp. 82–88.

[15] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE*. IEEE, 2009, pp. 485–495.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, USA, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[17] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: Patterns of cloning in software," *Empir. Softw. Eng.*, vol. 13, no. 6, pp. 645–692, 2008.

[18] C. Kästner and S. Apel, "Virtual separation of concerns – a second chance for preprocessors," *J. Object Technology*, vol. 8, no. 6, pp. 59–78, 2009.

[19] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*. ACM, 2008, pp. 311–320.

[20] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.

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

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

[23] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, 2006.

[24] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *ICSE*. ACM, 2010, pp. 105–114.

[25] R. E. Lopez-Herrejon and D. Batory, "A standard problem for evaluating product-line methodologies," in *GCSE*. Springer, 2001, pp. 10–24.

[26] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems," in *AOSD*. ACM, 2012, pp. 167–178.

[27] I. Macia Bertran, A. Garcia, and A. von Staa, "An exploratory study of code smells in evolving aspect-oriented systems," in *AOSD*, 2011, pp. 203–214.

[28] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

[29] N. Moha, "Detection and correction of design defects in object-oriented designs," in *OOPSLA*. ACM, 2007, pp. 949–950.

[30] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien, "A domain analysis to specify design defects and generate detection algorithms," in *FASE*. Springer, 2008, pp. 276–291.

[31] M. P. Monteiro and J. a. M. Fernandes, "Towards a catalogue of refactorings and code smells for AspectJ," *Trans. Aspect-Oriented Softw. Dev. I*, vol. 3880, pp. 214–258, 2006.

[32] T. Patzke, M. Becker, M. Steffens, K. Sierszecki, J. E. Savolainen, and T. Fogdal, "Identifying improvement potential in evolving product line infrastructures: 3 case studies," in *SPLC*. ACM, 2012, pp. 239–248.

[33] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *ECOOP*. Springer, 1997, pp. 419–443.

[34] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier, "Avoiding variability of method signatures in software product lines: A case study," in *GPCE Workshop on Aspect-Oriented Product Line Engineering*, 2007, pp. 20–25.

[35] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University at Kingston, Canada, Tech. Rep. 541, 2007.

[36] S. Schulze, S. Apel, and C. Kästner, "Code clones in feature-oriented software product lines," in *GPCE*. ACM, 2010, pp. 103–112.

[37] S. Schulze, E. Jürgens, and J. Feigenspan, "Analyzing the effect of preprocessor annotations on code clones," in *SCAM*. IEEE, 2011, pp. 115–124.

[38] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter? A controlled experiment," in *GPCE*. ACM, 2013, pp. 65–74.

[39] S. Schulze, O. Richers, and I. Schaefer, "Refactoring delta-oriented software product lines," in *AOSD*. ACM, 2013, pp. 73–84.

[40] D. C. Sharp, "Reducing avionics software cost through component based product line development," in *DASC*, vol. 2. IEEE, 1998, pp. G32/1–G32/8.

[41] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with C News," in *USENIX Tech. Conf.* USENIX association, 1992.

[42] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Sci. Comp. Prog.*, vol. 79, pp. 70–85, 2014.

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

[44] A. Yamashita, "How good are code smells for evaluating software maintainability? results from a comparative case study," in *ICSM*. IEEE, 2013, pp. 566–571.