Code Smells in Highly Configurable Software

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

***Abstract*—Modern software systems are increasingly configurable. *Conditional compilation* based on C preprocessor directives (i. e., `#ifdefs`) is a popular variability mechanism to implement this configurability in source code. Although C preprocessor usage has been subject to repeated criticism, with regard to variability implementation, there is no thorough understanding of which patterns are particularly harmful. Specifically, we lack empirical evidence of how frequently reputedly bad patterns occur in practice and which negative effect they have. For object-oriented software, in contrast, *code smells* are commonly used to describe source code that exhibits known design flaws, which negatively affect understandability or changeability. Established code smells, however, have no notion of variability. Consequently, they cannot characterize flawed patterns of variability implementation. The goal of my research is therefore to create a catalog of *variability-aware code smells.* I will collect empirical proof of how frequently these smells occur and what their negative impact is on understandability, changeability, and fault-proneness of affected code. Moreover, I will develop techniques to detect variability-aware code smells automatically and reliably.**

I. Introduction

Code smells are a concept to characterize source code that suffers from structural deficiencies that make it hard to understand, change, or test [1]. Fowler et al. introduced code smells as indicators that the source code structure might need to be improved through *refactoring*. Research has addressed the detection [2]–[4] and correction [5] of code smells. Moreover, the negative impact of code smells on software development has been studied [6]–[10]. Complementarily, Brown et al. have introduced *anti-patterns* [11], which are related to code smells but describe shortcomings with more profound consequences (e. g., architectural problems) and are not limited to code.

Despite the maturity of code smell and anti-pattern research for traditional software systems (especially object-oriented software), current approaches fall short when dealing with the variability of *highly configurable software systems*. A highly configurable software system (a. k. a. *software product line (SPL))* implements not just a single program, but a set of related programs (a *program family*), which are built from a common set of assets [12]–[14]. The commonalities and differences of members of this program family are communicated in terms of *features*, i. e., increments in functionality that are important to some stakeholder.

There are different *variability mechanisms* to implement highly configurable software systems. For instance, plug-in architectures, aspect-oriented programming, and feature-oriented programming have been proposed [15], [16]. In practice, however, *conditional compilation*, an *annotation-based* approach, is the predominant variability mechanism [17]. This mechanism follows a simple annotate-and-remove paradigm: All features are implemented in a common code base, and feature code is annotated in place, for instance, using `#ifdef` directives of the C preprocessor (CPP). Controlled by a configuration, feature code is conditionally excluded from compilation by means of a preprocessor.

I contend that variability adds a layer of complexity to highly configurable software that is not present in single software systems: Not only do classes, functions, etc. have to be structured, but the way that variability is implemented has to be structured as well. Hence, variability poses new design challenges, and some of the potential solutions arguably have more negative effects on the understandability and changeability of source code than others. Established code smells, however, have no notion of variability, and consequently fail to capture this difference in effects.

Based on my previous work [18], I therefore propose within my PhD research *variability-aware code smells*, specifically, variability-aware code smells for highly configurable software systems using preprocessor-based variability. Other researchers have proposed variability smells in a broader sense [14], [19]–[22], for instance, on the architectural level or for variability models. However, neither has source code been explicitly considered, nor has the harmfulness of these smells been shown empirically. Starting from established code smells and anti-patterns, I will derive a catalog of potential variability-aware code smells. I will then empirically validate this catalog by investigating which smells occur frequently in real-world systems and by gathering evidence that my proposed smells indeed negatively affect software development. As part of this empirical validation, I will develop techniques to automatically and reliably detect variability-aware code smells.

A. Research Questions

My objective is to investigate the relation between code smells and preprocessor-based variability mechanisms. To this end, I formulate the following research questions.

RQ 1 *How do annotation-based variability mechanisms affect the code smell concept?* The goal of this question is to derive a set of potential variability-aware code smells. I will investigate this topic with the following sub-questions. **RQ 1.1** *How do established code smells change when annotation-based variability is involved?* **RQ 1.2** *Are there additional variability-aware code smells, which are unrelated to existing code smells?*

RQ 2 *How can those variability-aware code smell be detected automatically?* By answering this question, I will gain a deeper understanding of the key properties of a particular code smell. Moreover, automatic detection will enable me to investigate how frequently a code smell occurs in practice.

RQ 3 *To which extent do proposed smells negatively affect the development of highly configurable software?* Specifically, I want to compare understandability, changeability and fault-proneness of code that suffers from variability-related code smells to code that does not suffer from these smells. To this end, I pose the following sub-questions. **RQ 3.1** *Is smelly code harder to understand (than non-smelly code)?* **RQ 3.2** *Is smelly code changed more frequently or more radically?* **RQ 3.3** *Is smelly code more prone to software faults?*

II. State of the Art

Similar to the foundational work of Fowler et al. [1], others have proposed code smells [23]–[29]. Some of this work also proposed smell detection techniques [26]–[28]. However, these smells either do not address variability [23], [24], or concentrate on aspect-oriented or delta-oriented programming [25]–[29]. In contrast, I am interested in smells related to source-code variability, and focus on preprocessor-based variability.

Apel et al. presented a summary of fourteen *variability smells* for SPLs [14]. Similarly, based on practical experience, others have reported recurring variability-related problems [19]–[22]. Differently from my work, these smells and problems are not limited to code, but also target other phases of SPL engineering, such as variability modeling or product derivation. Although these smells are founded on expert knowledge, they require further, detailed investigation.

Based on problems in evolving SPLs, Patzke et al. analyze root causes of those problems by means of expert interviews and analysis of a case study [30]. Some of the identified root causes may manifest as code smells. Thus, there is synergy potential between his research and mine.

More recently, Abilio et al. proposed three code smells, as well as detection metrics for SPLs that use feature-oriented programming [31]. Their smells are derived from well-known object-oriented smells, whose underlying problems (e. g., scattering, tangling) are also important for software with preprocessor-based variability. However, in code with preprocessor variability, these problems will manifest differently, and consequently, require different detection strategies.

Further research has used SPLs as subjects for the detection of traditional code smells and architectural smells [32]–[34]. Carneiro et al. have proposed concern-sensitive visualizations to help identify well-known object-oriented smells [35]. Furthermore, Figueiredo et al. have proposed a set of concern-sensitive heuristics to detect modularity flaws [36]. Their work investigates known object-oriented and aspect-oriented smells, putting an emphasis on crosscutting concerns. I, in contrast, am interested in code smells that are specifically related to variability and focus on preprocessor-based variability.

The smell Duplicated Code (a. k. a. code clones), has been investigated in the context of highly configurable systems [37], [38]. While not neglecting Duplicated Code entirely, I will also investigate other smells.

Finally, a large body of work deals with shortcomings of the CPP. Among others, the negative impact of CPP directives on understandability, changeability, and error-proneness has been discussed [39]–[41]. With respect to variability, Liebig et al. present empirical results on the usage of `#ifdef` directives, specifically with a focus on granularity, scattering, tangling, and nesting of such directives [17]. While this work is related due to the focus on CPP use, it does not provide concrete patterns of misuse. In contrast, I will investigate concrete smells and relate them to problems for program comprehension and maintainability.

III. Research Methodology

In this section, I outline how I will investigate and evaluate the questions presented in Section I-A.

RQ 1: In order to answer how preprocessor-based variability affects code smells, I will develop a set of candidate variability-aware code smells. Research questions 2 and 3, in turn, are specifically designed to evaluate the occurrence of these candidate smells in practice (RQ 2), and their negative impact on software development (RQ 3), respectively.

There are two sources of potential smells. First, I will derive possible variability-aware code smells from established code smells and anti-patterns by applying the methodology described in my previous work [18]. The methodology works by systematically introducing variability into the code pattern associated with a particular smell. For instance, the established smell Long Method characterizes a method that implements too much functionality, which typically manifests as a method with a large number of statements [1]. To make this smell variability-aware, I consider what happens to a long method when many of its statements are annotated with preprocessor directives. The result is a heavily annotated method, which I argue would be just as problematic as its object-oriented counterpart, but for different reasons.

As a second source of potential smells, I will solicit proposals from practitioners and fellow researchers, for instance, by conducting interviews and surveys. Examples of smelly code suggested in this manner will be collected and discussed in an open database, similar to the effort by Palomba et al. for object-oriented code smells [42] or Abal et al. for variability-related bugs [43].

RQ 2: Inspired by the DECOR framework [4], I plan to provide a configurable smell detection framework that allows for the flexible combination of different metrics in order to detect instances of variability-aware code smells. Part of these metrics (e. g., scattering and tangling), will be determined by analyzing the static structure of source code. Other metrics, such as code churn or co-change dependencies, can be better detected by analyzing the evolution of a subject system [44]. I will extract these metrics by mining version control histories of subject systems. The accuracy of the tool will be validated by measuring its performance in terms of precision and recall against a manually validated set of code smell instances.

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

Fig. 1. A heavily annotated function in MySQL. I propose the variability-aware code smell ANNOTATION BUNDLE to describe functions annotated in this manner.

RQ 3: I will investigate the negative impact of potential variability-aware code smells empirically. Understandability and changeability (RQ 3.1 and 3.2) of smelly code will be rated using questionnaires and experiments. These ratings will be complemented with expert interviews with the original developers of the subject systems. Change-proneness of smelly code (RQ 3.2) will additionally be investigated by mining version control information. Furthermore, I will combine version control and bug tracking information in order to establish whether or not smelly code is more fault-prone than non-smelly code (RQ 3.3). This combination was previously employed by Khomh et al. to investigate change- and fault-proneness of code suffering from object-oriented smells [9].

IV. KEY PRELIMINARY RESULTS

In previous work, I have introduced the notion of variability-aware code smells and a methodology to derive variability-aware code smells from traditional, single-system smells [18]. Furthermore, I presented an initial catalog of four smells for different variability mechanisms. This catalog was validated with the help of a survey among researchers in the field of highly configurable software systems.

As an example of a variability-aware code smell, I show in Fig. 1 a function that suffers from the ANNOTATION BUNDLE smell, which was derived from the object-oriented smell LONG METHOD [1]. This function was extracted from the open-source database management system MySQL, version 5.6.17, file `mysys/thr_alarm.c`.[1] Although it is not especially long in terms of lines of code, it contains a high amount of

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

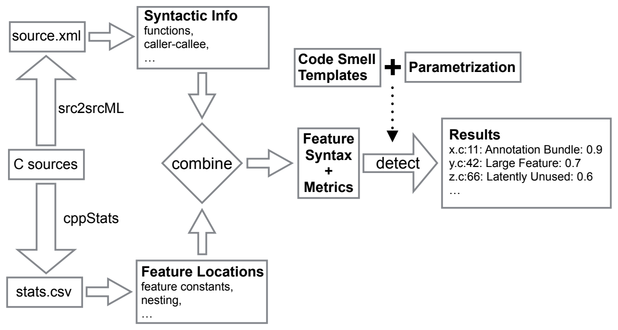

Fig. 2. Preliminary architecture of my proposed variability-aware code smell detection tool

variability, which is due to the high number of CPP annotations (starting on Lines 6, 10, 15, 21, and 22). Because of these annotations, this function can be compiled in a large number of variants, all of which must be considered when trying to understand, change, or test it. I argue that a function that is annotated in this manner is hard to understand and change, and therefore propose this pattern as a code smell.

In Fig. 2, I show the preliminary architecture of the variability-aware code smell detection tool for C code containing CPP annotations, which I extend and refine in ongoing work. Detection starts with a set of C source code files, which are analyzed by two external tools, CPPSTATS[2] and SRC2SRCML[3] [45]. These tools extract syntactic information (e. g., locations of function definitions), and information related to annotations (e. g., the number of preprocessor macros). My tool then combines this information and applies parametrized *code smell templates* in order to detect code sections that suffer from smells. The result of the detection process is a list of locations that may be infected with a smell, along with a numeric value indicating the severity of infection.

Preliminary runs of the detection tool on a number of open-source subjects (e. g., OpenVPN, Vim, libxml2) indicate that my proposed variability-aware code smell ANNOTATION BUNDLE frequently occurs in practice. Moreover, the number of smell occurrences varies widely from subject to subject.

V. CONCLUSION

Code smells are an established concept to identify flawed solutions to recurring problems. Existing code smells are geared at software whose source code structure is essentially fixed. However, complex software systems are increasingly highly configurable, meaning that they can be customized for different operating systems, hardware platforms, and so on. This configurability manifests in the form of variable source code, for instance, in source code that is annotated with C preprocessor directives. I argue that variable source code adds a new layer of complexity that is not present in single software systems. Some of the possible solutions to deal with this complexity have a potentially detrimental effect on the

[2]http://www.fosd.net/cppstats/
[3]http://srcml.org/

understandability or changeability of source code. Established code smells, however, are ill-equipped to characterize these problematic solutions.

The goal of my research is therefore to make code smells variability-aware. Due to their wide-spread use, I will focus on preprocessor-based variability mechanisms. Based on established code smells, I will derive a set of candidate variability-aware code smells, investigate which of those smells occur frequently in practice, and whether or not they have a negative impact on software development. The envisioned contribution of my research will be a catalog of variability-aware code smells that I hope will raise awareness to recurring variability-related problems. This awareness may in turn help increase the internal quality of highly configurable software systems.

ACKNOWLEDGMENTS

I thank Gunter Saake for advising me on this thesis topic. Moreover, I thank Sandro Schulze and Thomas Thüm for their collaboration and numerous fruitful discussions.

REFERENCES

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

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

[3] 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.

[4] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the Specification and Detection of Code and Design Smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

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

[6] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

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

[8] 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.

[9] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empir. Softw. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.

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

[11] W. H. Brown, R. C. Malveau, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

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

[13] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

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

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

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

[17] 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.

[18] W. Fenske and S. Schulze, "Code smells revisited: A variability perspective," in *VaMoS*. ACM, 2015, pp. 3–10.

[19] P. G. Bassett, *Framing Software Reuse: Lessons from the Real World*. Prentice-Hall, Inc., 1997.

[20] C. W. Krueger, "The 3-tiered methodology," in *SPLC*. IEEE, 2007, pp. 97–106.

[21] ——, "New methods behind a new generation of software product line successes," in *Applied Software Product Line Engineering*, K. C. Kang, V. Sugumaran, and S. Park, Eds. CRC Press, 2009.

[22] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "Refactoring a legacy component for reuse in a software product line: A case study," *Softw. Maint.: Res. Pract.*, vol. 18, no. 2, pp. 109–132, Apr. 2006.

[23] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2004.

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

[25] M. P. Monteiro and J. a. M. Fernandes, "Towards a catalogue of refactorings and code smells for AspectJ," in *Trans. Aspect-Oriented Softw. Development I*. Springer, 2006, pp. 212–258.

[26] E. K. Piveta, M. Hecht, M. S. Pimenta, and R. T. Price, "Detecting bad smells in AspectJ," *J. Univ. Comput. Science*, vol. 12, no. 7, pp. 811–827, 2006.

[27] K. Srivisut and P. Muenchaisri, "Bad-smell metrics for aspect-oriented software," in *Int'l Conf. on Computer and Information Science (ICIS)*. IEEE, 2007, pp. 1060–1065.

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

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

[30] 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.

[31] R. Abilio, J. Padilha, E. Figueiredo, and H. Costa, "Detecting code smells in software product lines – an exploratory study," in *Int'l Conf. on Information Technology - New Generations (ITNG)*. IEEE, 2015, pp. 433–438.

[32] 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.

[33] E. Guimaraes, A. Garcia, E. Figueiredo, and Y. Cai, "Prioritizing software anomalies with software metrics and architecture blueprints," in *Int'l Work. on Modeling in Software Engineering (MiSE)*. IEEE, 2013, pp. 82–88.

[34] 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.

[35] G. de F Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonça, "Identifying code smells with multiple concern views," in *Brazilian Symposium on Software Engineering (SBES)*. IEEE, 2010, pp. 128–137.

[36] E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena, "Applying and evaluating concern-sensitive design heuristics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 227–243, 2012.

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

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

[39] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with C News," in *Proc. USENIX Conf.* USENIX Association, 1992, pp. 185–197.

[40] 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, Dec. 2002.

[41] 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.

[42] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Landfill: An open dataset of code smells with public evaluation," in *MSR (Data Papers Track)*. IEEE, 2015, p. 4.

[43] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the Linux kernel: A qualitative analysis," in *ASE*. ACM, 2014, pp. 421–432.

[44] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 462–489, 2015.

[45] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An XML-based lightweight C++ fact extractor," in *IWPC*. IEEE, 2003, pp. 134–143.