Code Smell Analysis in Cloned Java Variants: the Apo-games Case Study

Luciano Marchezan
ISSE, Johannes Kepler University Linz
Linz, Austria

Wesley K. G. Assunção
ISSE, Johannes Kepler University Linz
Linz, Austria

Gabriela Michelon
ISSE, Johannes Kepler University Linz
Linz, Austria

Edvin Herac
ISSE, Johannes Kepler University Linz
Linz, Austria

Alexander Egyed
ISSE, Johannes Kepler University Linz
Linz, Austria

ABSTRACT

Families of software products are usually created using opportunistic reuse (*clone-and-own*) in which products are cloned and adapted to meet new requirements, user preferences, or non-functional properties. Opportunistic reuse brings short-term benefits, e.g., reduced time-to-market, whereas creating long-term drawbacks, e.g., the need of changing multiple variants for any maintenance and evolution activity. This situation is even worse when the individual products have poor design or implementation choices, the so-called code smells. Due to their harmfulness to software quality, code smells should be detected and removed as early as possible. In a family of software products, the same code smell must be identified and removed in all variants where it is are present. Identifying instances of similar code smells affecting different variants has not been investigated in the literature yet. This is the case of the Apo-Games family, which has the challenge of identifying the flaws in the design and implementation of cloned games. To address this challenge, we applied our inconsistency and repair approach to detect and suggest solutions for six types of code smells in 19 products of the Apo-games family. Our results show that a considerable number of smells were identified, most of them for the long parameter list and data class types. The number of the same smells identified in multiple variants ranged between 2.9 and 20.2 on average, showing that *clone-and-own* may lead to the replication of code smells in multiple products. Lastly, our approach was able to generate between 4.9 and 28.98 repair alternatives per smell on average.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Maintaining software**.

KEYWORDS

software product line, code smells, consistency checking, inconsistency repair

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '22, September 12–16, 2022, Graz, Austria

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9443-7/22/09…$15.00
https://doi.org/10.1145/3546932.3547015

ACM Reference Format:
Luciano Marchezan, Wesley K. G. Assunção, Gabriela Michelon, Edvin Herac, and Alexander Egyed. 2022. Code Smell Analysis in Cloned Java Variants: the Apo-games Case Study. In *26th ACM International Systems and Software Product Line Conference - Volume A (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3546932.3547015

1 INTRODUCTION

Software Product Line (SPL) [17] is a well-established approach to systematically dealing with a family of software products. However, designing and implementing an SPL is a complex task [11]. Thus, the most common industrial practice for creating families of software products is by the adoption of opportunistic reuse [1], also known as *clone-and-own* [2]. When companies use opportunistic reuse, new software products are created by cloning existing software products and adapting them with the desired changes. Eventually, the family of software products is re-engineered into an SPL [1], or maintained product-by-product [6]. Nonetheless, as it happens for traditional single-product software, software products created using opportunistic reuse also suffer from poor design and/or implementation choices, the so-called code smells [4]. Code smells make software systems hard to evolve and maintain and can be harmful as they potentially lead to bugs or difficult program comprehension. In the case of a family of cloned systems, the existence of code smells is even worse, as such flaws are propagated to several products [7]. Hence, identifying and removing such code smells is a challenging activity, requiring analysis of multiple products and propagation of correction changes.

Families of software products, as well as code smells, are two research topics that are widely investigated individually, but have not been investigated together. The most related studies investigate variability smells [3, 14] but not in the context of opportunistic reuse, or variability debt [18] but not in the context of code smells. This means that there is a gap in the literature, which is highlighted in the work of Krüger et al. [7], introducing a challenge of "Code Smell Analysis" for the Apo-Games case study.

There are different approaches proposed to identify code smells [5, 12, 13, 15] that could be applied to cloned variants. These approaches, however, are limited to focusing on specific code smells, only applicable to specific artifacts (only source code), or do not recommend fixes for the smells identified. Based on these limitations, in this paper, we apply an inconsistency detection approach [10] to identify code smells in the Apo-Games variants. Our approach is customizable by the use of Consistency Rules (CR) for finding the

smells. Variations of the approach were applied in different types of artifacts [8, 10, 16], however, in this paper, we investigate how our approach can be applied in product variants developed using *clone-and-own*. Based on the code smells identified, our approach generates repair alternatives for fixing them.

The results of the proposed solution were obtained by applying our approach to identify six types of code smells in 13 Java and 6 Android products from the Apo-games family. The results show that our approach was able to identify between 5 and 145 smells per system. The smells identified, appeared multiple times, with the most recurrent smell appearing 49 times across the game variants. Considering the recurrence of smells between the systems, we observed that *ApoSimple* is the game that shares more smells with the other products, 20.2 on average with each product. Furthermore, the results also indicated that Android-based games share at least more than 10 smells with a specific group of Java products (*ApoSimple*, *ApoSimpleSud*, and *ApoSlitherLink*). Considering the repairs generated, the approach generated between 4.9 and 29.98 repair alternatives on average per product.

2 METHODOLOGY

To collect the data for this solution paper, we applied our inconsistency detection and repair approach [10]. The key definitions of our approach are described next:

Definition 1 - System: consists of elements which contain properties. A property of an element is referred to by element dot (.) property name, e.g., "Car.name". A property can be of a primitive type (e.g., Boolean, Integer, Float, or String) or a reference to other elements. Hence, elements are instances of a specific type.

Definition 2 - Consistency Rule (CR): a condition defined for a context that must be fulfilled by an element. This condition evaluates to a Boolean value as *true* (consistent) or *false* (inconsistent). A consistency rule is defined for a context. The context is a meta element, which is a type of a element (e.g., class or method).

Definition 3 - Repair: a non-empty set of repair actions that fixes a specific inconsistency from the set of all possible inconsistencies. A repair action identifies the operation, element, element property, and, optionally, a concrete value to change the element property. The following operations are possible: **add** an element to the system or to a collection of elements, **delete** (del) an element from the system or from a collection, and **modify** (mod) an element property to a given value.

Definition 4 - Repair Tree: a hierarchically ordered set of repair nodes for a single inconsistency. The nodes of a repair tree define whether the underlying repairs are alternatives (*) or sequences (+). Repair alternative nodes follow the exclusive or-alternative (XOR) principle where, from a set of repair alternatives, only one must be selected for fixing an inconsistency. An example of a repair tree is illustrated in Figure 1. This repair tree is created by applying a CR (CR1) in an element called "play", which represents a method in a class called Streamer. CR1 states that methods can not have the same signature, i.e., the same name and arguments. The method *play*, however, appears twice in the class with the same signature. The repair tree, suggests six alternative repairs (*): add a new argument to the first method *play*, add a new argument to the second method *play*, delete the first method *play*, delete the second method *play*, modify the name of the first method *play*, or modify the name of the second method *play*. As these are repair alternatives, executing one of those will fix the inconsistency for CR1.

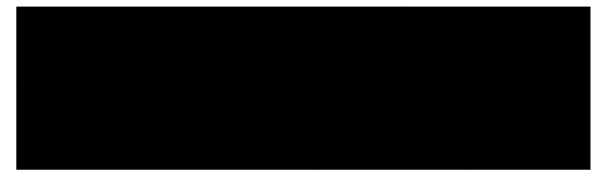

Figure 1: Repair tree example

In the context of this work, we defined six CRs, which are used to identify inconsistencies based on six types of code smells. Table 1 presents the six CRs. We defined specific metrics for considering a smell, such as CR1 (in Table 1) that says a class with more than 20 fields is considered a *Large Class*. This number, however, was defined based on our agreement relating to how many fields a class may have until it becomes a smell. Such a value may vary depending on the developer's preferences. This value directly impacts the results of the approach, as if the value was 10 instead of 20, probably more smells would be identified. However, this is not an issue as the CRs could be easily changed based on user preferences, allowing our approach to be customizable for different contexts. Moreover, our goal is not to obtain a unique result regarding the code smells found in the variants, but rather to collect data about the smells to analyze the applicability of our approach as well as the potential relationship between the smells and the variants' similarities/differences. After the identification of the code smells, our approach also generated repair alternatives that could be used for fixing the smells. This repair generation is based on the approach presented by [10]. The CRs definition is given using the Abstract Rule Language (ARL),[1] such definitions are not given in Table 1 due to space limitation.[2] The six CRs were applied in 13 Java and six Android products of the Apo-games family.

Our inconsistency detection and repair approach [10] was implemented as part of the DesignSpace research project.[3] The prototype implementation is a service that runs on a server. This server can be accessed by different engineering tools through the use of plugins. These plugins work as a *middleware* that connect engineering tools to the server. DesignSpace includes plugins for tools such as IntelliJ, Eclipse, Visio, SolidWorks, and EclipsePapyrus, among others. To analyze the code from the Apo-Games, we used the plugin for IntelliJ IDEA, which allows us to connect the IDEA to the DesingSpace server, where our approach is running as a service. Once the IDEA is connected, data about the project's source code can be analyzed. This data includes, but is not limited to: java files, classes, fields, methods, and statements. Once this data is available on the server, our approach can apply the CRs defined to find inconsistencies.[4]

[1] ARL documentation available at: https://isse.jku.at/designspace/index.php/Abstract_Rule_Language
[2] CRs definitions are available at our online repository [9]
[3] Project documentation available at: https://isse.jku.at/designspace/
[4] A demo video showing how to use our approach to detect smells is available at the DesignSpace wiki: https://isse.jku.at/designspace/index.php/Demos

Table 1: Consistency rules used to identify smells

ID	Smell	Definition
CR1	Large Class	Class has more than 20 fields
CR2	Large Class	Class has more than 20 methods
CR3	Long Methods	Method has more than 20 statements
CR4	Long Parameter List	Method has more than 4 parameters
CR5	Primitive Obsession	Class has more than 10 primitive fields
CR6	Data Class	Class contains only crude methods

Table 2: Summary of consistency checking per system

System	Size	Files	Evaluations	Smells
Java				
ApoSimple	40362	112	4377	102
ApoComm.	30314	78	3240	96
ApoDefence	27458	69	2537	85
ApoRelax	16136	56	2099	70
ApoPongBeat	12560	79	2497	67
ApoNotSoSim.	15042	57	2223	63
ApoSlitherLink	15571	62	2564	59
ApoIcarus	12577	59	1933	59
ApoMarc	11391	60	2028	51
ApoSimpleSud.	11087	43	1629	49
ApoBot	13537	48	1881	47
ApoStarz	14447	49	1913	44
TutorVolley	427	6	208	5
Android				
BitsEngine	20974	57	2699	145
ApoMono	12727	24	1256	46
myTreasure	9410	27	1122	30
ApoClock	7217	28	1136	30
ApoSnake	6322	19	814	26
ApoDice	5274	19	823	22

Table 3: Summary of consistency checking per CR

System	Smells Count					
	CR1	CR2	CR3	CR4	CR5	CR6
Java						
ApoBot	3	8	4	24	0	8
ApoComm.	4	14	12	54	0	12
ApoDefence	7	10	14	42	0	12
ApoIcarus	2	7	4	36	0	10
ApoMarc	1	7	2	31	0	10
ApoNotSoSim.	3	11	3	35	0	11
ApoPongBeat	2	8	6	38	0	13
ApoRelax	3	9	11	37	0	10
ApoSimple	7	16	10	56	0	13
ApoSimpleSud.	1	7	3	28	0	10
ApoSlitherLink	3	9	8	29	0	10
ApoStarz	3	7	3	22	0	9
TutorVolley	0	0	0	0	0	5
Android						
ApoClock	0	2	3	19	0	6
ApoDice	0	2	1	12	0	7
ApoMono	2	4	5	30	0	5
ApoSnake	0	2	4	13	0	7
BitsEngine	5	8	6	114	0	12
myTreasure	3	3	5	12	0	7

Table 4: Smells recurrence grouped per CR

CR	Min	Max	Mean	Median	Std
CR1	1	1	1	1	0
CR2	1	12	1.38	1	1.32
CR3	1	12	1.67	1	1.95
CR4	1	49	2.70	1	4.94
CR6	1	15	2.95	1	4.01

3 RESULTS AND DISCUSSION

Table 2 shows a summary of the application of our approach in the 19 game variants. Column *size* describes the number of elements extracted from the system into our server (files, classes, methods, fields, statements). Column *evaluations* represents the number of times that a CR was applied to the system's element. For instance, CR1 (Table 1) was applied in all classes in each system. Column *smells* shows the total number of smells identified, i.e., the number of evaluations that resulted to *false* (see Definition 2). The number of smells identified (Table 2) ranged from 5 (TutorVolley) to 102 (ApoSimple) in the Java products and from 22 (ApoDice) to 145 (BitsEngine) in the Android products. As shown in the results, the number of smells, in most cases, is impacted by the number of elements, i.e., a bigger number of smells were identified in bigger systems. Thus, our results lead us to conclude that:

Conclusion 1: *Our approach can be used to identify code smells on product variants. The number of smells identified is in most cases impacted by the size of the product.*

Table 3 shows results related to the smells identified per rule in each system. CR5 was the only rule applied in which smells were never found. For the other rules, CR4 (Long Parameter List) and CR6 (Data Class) smells were the most common. Table 4 shows the results regarding the recurrence of the same smell per CR. As described, the smell that appeared more times was related to CR4, appearing 49 times (column *max*) across the systems. Furthermore, CR6 had the same smells identified 2.95 times on average (column *Mean*). This shows that the smells found were not unique (except for CR1), indicating that opportunistic reuse can duplicate the smells across the system and across multiple systems. This conclusion is further supported by the results shown in Table 5. These results show how many smells were found between the systems. For instance, *ApoBot* and *ApoComm.* have 27 smells in common. The system that contained the most recurrent smells was *ApoSimple*, sharing between four and 43 with all other systems, 20.21 on average. Another interesting result is the number of shared smells considering the Android products as they share a few to zero smells with most Java products, except for *ApoSimple*, *ApoSimpleSud.*, and *ApoSlitherLink*. The number of smells shared only among the Android products is higher, as expected, as most of their code should be similar considering their platform. Hence, we conclude that:

Conclusion 2: *Opportunistic reuse may lead to a recurrence of the same smells across multiple products.*

The results presented in Table 5 are also important to understand the effort required to fix the smells. For example, *ApoIcarus* and *ApoMarc* have 14 smells in common. Thus, the fixes applied in these 14 smells in *ApoIcarus* could be propagated into *ApoMarc* to fix the same 14 smells. Our approach was able to generate fixes (repairs) as shown in Table 6. The number of repair alternatives generated per smell ranged from 1 (TutorVolley) to 29.98 (ApoDefence) on average per Java product. For Android products, the average number ranged from 6.76 to 13.86 per smell. The average number of repair alternatives was similar for systems that share more smells. For

Table 5: Smells recurrence between products

Systems	Java Systems													Android Systems							
	ApoBot	ApoComm.	ApoDefence	ApoIcarus	ApoMarc	ApoNotSoSim.	ApoPongBeat	ApoRelax	ApoSimple	ApoSimpleSud.	ApoSlitherLink	ApoStarz	TutorVolley	ApoClock	ApoDice	ApoMono	ApoSnake	BitsEngine	myTreasure	Mean	Std.
ApoBot		27	7	16	-	-	-	7	27	26	33	28	4	9	9	8	8	5	10	14.3	13.2
ApoComm.			9	25	-	-	-	9	40	37	32	29	4	9	11	9	10	7	10	19.2	22.5
ApoDefence				-	-	-	-	8	11	9	10	7	4	5	7	5	6	7	7	9.8	18.5
ApoIcarus					14	15	14	9	21	20	16	15	-	2	2	2	2	1	2	12.4	13.9
ApoMarc						32	35	28	-	-	-	-	-	-	-	-	-	-	-	8.4	15.8
ApoNotSoSim.							35	26	-	-	-	-	-	-	-	-	-	-	-	9	17.6
ApoPongBeat								30	-	-	-	-	-	-	-	-	-	-	-	9.5	18.7
ApoRelax									9	9	9	8	4	4	6	5	6	7	6	13.7	15.8
ApoSimple										38	43	29	4	10	11	10	11	7	11	20.2	24
ApoSimpleSud.											33	28	4	11	13	11	12	7	10	16.7	14.4
ApoSlitherLink												32	4	9	10	10	11	7	10	17.3	16.5
ApoStarz													4	7	8	8	8	6	9	14.2	13.1
TutorVolley														3	3	3	3	3	4	2.9	1.6
ApoClock															20	22	21	5	20	9.8	8.7
ApoDice																16	17	6	13	9.2	6.6
ApoMono																	26	6	20	10.9	11.3
ApoSnake																		6	15	9.9	8.1
BitsEngine																			18	12.8	32.3
myTreasure																				10.3	7.9

instance, *ApoStarz* has 44 smells in total, in which 32 are shared with *ApoSlitherLink* (Table 5). This is an indication to the reason the number of repairs alternatives generated (Table 6) for these products is similar: they have the same median (7), similar average (11.71 for *ApoSlitherLink* and 12.36 for *ApoStarz*), and similar standard deviation (15.58 for *ApoSlitherLink* and 14.94 for *ApoStarz*). This indicates that repairs applied for one system may be reused for others, reducing the effort of fixing the smells and propagating the repairs. Further study regarding the change propagation necessary for fixing the smells will be investigated in the future. Furthermore, our approach provided a varied number of repair alternatives. This is important as it gives developers flexibility when deciding how to fix the smells. The results considering the repair generation lead us to conclude that:

Conclusion 3: *Recurrence of code smells may bring benefits considering how to repair them, as similar repairs may be reused for products sharing the same smells.*

4 THREATS TO VALIDITY

Internal Validity: An internal threat is the set of consistency rules used to identify the smells. To mitigate this threat, we applied rules to identify smells of different types considering different elements. When considering the metrics used in the rules (e.g., number of maximum parameters for CR4) we defined them based on our experience with software projects, as well as the suggestions found in code smells catalogs.[5]

External Validity: An external threat is related to the generalization of our results to another family of products. The results obtained indicate how smells are created and shared among variants created based on opportunistic reuse. However, as the Apo-Games products are varied both in size and complexity, we argue that other products developed by applying opportunistic reuse might present similar results. Furthermore, the Apo-Games is a publicly available data-set of product variants created with *clone-and-own*. Other data sets similar to this are not easily found.

[5]One catalog is present in Refactoring Guru: https://refactoring.guru/

Table 6: Repair generation results

System	Number of Repair Alternatives				
	Min	Max	Mean	Median	Std
Java					
ApoBot	1	67	11.25	5	15.11
ApoComm.	1	215	12.79	7	25.59
ApoDefence	1	715	29.98	5	100.46
ApoIcarus	1	123	9.33	5	17.29
ApoMarc	1	49	6.29	5	7.37
ApoNotSoSim.	1	69	9.44	7	11.80
ApoPongBeat	1	59	8.64	5	10.97
ApoRelax	1	543	16.8	7	64.46
ApoSimple	1	351	18.50	5	48.82
ApoSimpleSud.	1	49	7.77	7	8.21
ApoSlitherLink	1	81	11.71	7	15.58
ApoStarz	1	49	12.36	7	14.94
TutorVolley	1	1	1	1	0
Android					
ApoClock	1	69	7.53	3	14.48
ApoDice	1	43	4.90	3	8.73
ApoMono	1	115	13.86	5	24.17
ApoSnake	1	43	6.76	3	10.91
BitsEngine	1	419	12.82	5	39.81
myTreasure	1	71	11.66	3	17.29

5 CONCLUSION AND FUTURE WORK

In this paper, we present the first challenge solution related to the identification of code smells across product variants created with opportunistic reuse [7]. This analysis is important as it can aid the understanding of how the use of *clone-and-own* can spread the same smells across multiple variants. To collect the data related to code smells, we applied our inconsistency detection and repair approach [10] using six CRs into a set of 19 products from the Apo-games family. The results show that the most recurrent smell appeared 49 times across one or more products on average. Furthermore, the number of smells that were shared between two products ranged from 2.9 to 20.2 on average. This indicates that the application of opportunistic reuse may decrease the code quality of the whole family, as problems are being cloned into other products. However, this also indicates that the effort for fixing these problems can be reduced, as repairs created to fix smells on one system may be reused in other systems that have the same smells. Furthermore, our approach allows flexibility when fixing the smells as the number of repair alternatives ranged between 4.9 and 28.98 per smell on average. For future work, we plan to investigate the execution of the repairs, applying a change propagation approach [8] that may give us indications related to the effort required to fix code smells among multiple products considering the reusability of the fixes.

ACKNOWLEDGMENTS

The research reported in this paper has been partly funded by the Austrian Science Fund (FWF) (grant # P31989-N31 as well as grant # I4744-N) and by the Austrian COMET K1-Centre Pro2Future of the Austrian Research Promotion Agency (FFG) with funding from the Austrian ministries BMVIT and BMDW.

REFERENCES

[1] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (feb 2017), 2972–3016. https://doi.org/10.1007/s10664-017-9499-z

[2] Jorge Echeverría, Francisca Pérez, José Ignacio Panach, and Carlos Cetina. 2021. An empirical study of performance using Clone & Own and Software Product Lines in an industrial context. *Information and Software Technology* 130 (2021), 106444.

[3] Wolfram Fenske and Sandro Schulze. 2015. Code Smells Revisited: A Variability Perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-Intensive Systems* (Hildesheim, Germany) *(VaMoS '15)*. Association for Computing Machinery, New York, NY, USA, 3–10.

[4] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[5] Mouna Hadj-Kacem and Nadia Bouassida. 2018. A Hybrid Approach To Detect Code Smells using Deep Learning.. In *ENASE*. 137–146.

[6] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 432–444. https://doi.org/10.1145/3368089.3409684

[7] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-Games: A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 251–256. https://doi.org/10.1145/3233027.3236403

[8] Luciano Marchezan, Wesley K. G. Assuncao, Roland Kretschmer, and Alexander Egyed. 2022. Change-Oriented Repair Propagation. In *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering* (Pittsburgh, PA, USA) *(ICSSP'22)*. Association for Computing Machinery, New York, NY, USA, 82–92. https://doi.org/10.1145/3529320.3529330

[9] Luciano Marchezan, Wesley K. G. Assunção, Gabriela Michelon, Edvin Herac, and Alexander Egyed. 2022. *Applying an Inconsistency Repair Mechanism for clone-and-own Code Smell Analysis: the Apo-games Case Study (Evaluation Data)*. https://doi.org/10.5281/zenodo.6617601

[10] Luciano Marchezan, Roland Kretschmer, Wesley KG Assunção, Alexander Reder, and Alexander Egyed. 2022. Generating repairs for inconsistent models. *Software and Systems Modeling* (2022), 1–33.

[11] Luciano Marchezan, Elder Rodrigues, Wesley Klewerton Guez Assunção, Maicon Bernardino, Fábio Paulo Basso, and João Carbonell. 2022. Software product line scoping: A systematic literature review. *Journal of Systems and Software* 186 (apr 2022), 111189. https://doi.org/10.1016/j.jss.2021.111189

[12] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. 2016. JDeodorant: Clone Refactoring. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 613–616. https://doi.org/10.1145/2889160.2889168

[13] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2014. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* 41, 5 (2014), 462–489.

[14] Iuri Santos Souza, Ivan Machado, Carolyn Seaman, Gecynalda Gomes, Christina Chavez, Eduardo Santana de Almeida, and Paulo Masiero. 2019. Investigating Variability-Aware Smells in SPLs: An Exploratory Study. In *33rd Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES 2019)*. Association for Computing Machinery, New York, NY, USA, 367–376. https://doi.org/10.1145/3350768.3350774

[15] Amjed Tahir, Aiko Yamashita, Sherlock Licorish, Jens Dietrich, and Steve Counsell. 2018. Can You Tell Me If It Smells? A Study on How Developers Discuss Code Smells and Anti-Patterns in Stack Overflow. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand) *(EASE'18)*. Association for Computing Machinery, New York, NY, USA, 68–78. https://doi.org/10.1145/3210459.3210466

[16] Michael Alexander Tröls, Luciano Marchezan, Atif Mashkoor, and Alexander Egyed. 2022. Instant and global consistency checking during collaborative engineering. *Software and Systems Modeling* (2022), 1–27.

[17] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media.

[18] Daniele Wolfart, Wesley K. G. Assunção, and Jabier Martinez. 2021. Variability Debt: Characterization, Causes and Consequences. In *20th Brazilian Software Quality Symposium (SBQS)*.