2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 978-1-6654-9598-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-Companion55297.2022.9793758

Completeness of Composite Refactorings for Smell Removal

Ana Carla Bibiano
abibiano@inf.puc-rio.br
Opus Research Group, Informatics Department, PUC-Rio
Brazil

ABSTRACT

Code smells are problems in the internal structural quality. Refactoring is a technique commonly used to remove code smells. A single refactoring rarely suffices to assist developers in achieving a full removal of a code smell. Thus, developers frequently apply composite refactorings (or, simply, composites) with the goal of fully removing a smell. A composite is formed by two or more interrelated single refactorings. Studies report that developers often fail in fully removing code smells through composite refactoring. In this context, a composite refactoring is considered incomplete whenever it does not fully remove a smell. Either incomplete or complete composite is formed by several refactorings; thus, both may inadvertently degrade other parts of the software. However, the literature on (in)complete composites and their effects on structural quality is still scarce. This lack of knowledge hampers the design of empirically-based recommendations to properly assist developers in performing effective complete composites, i.e., those not causing any harm in related parts of the program. This doctoral research investigates the effect of composite (in)completeness on structural quality and proposes a catalog with composite recommendations for the full removal of popular code smell types. We investigated 618 composites in 20 software projects. We found that 58% of incomplete composites did not change the internal structural quality, and 64% of complete composites are formed by refactoring types that were not actually previously recommended in the literature or elsewhere. The expected contributions are a list of findings, guidelines, and a catalog to support developers on how to successfully perform complete composites.

KEYWORDS

Refactoring, Composite refactoring, Composite Completeness, Code smells

ACM Reference Format:
Ana Carla Bibiano. 2022. Completeness of Composite Refactorings for Smell Removal. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3517060

1 INTRODUCTION

Refactoring is a code transformation applied to improve the internal structural quality [15, 19]. A single *refactoring* is an instance of a refactoring *type*. Each refactoring type partially or fully contributes to the full removal of poor code structures [4, 9], such as *code smells* [14, 30]. Example of code smells are *Feature Envy* or a *God Class*. Given its fine-grained nature, a single refactoring rarely suffices to fully remove even a quite simple smell [9, 14, 30]. Developers then frequently apply *composite refactorings* or, simply, *composites* [4, 11] aiming to fully remove code smells. A composite refactoring is formed by two or more interrelated single refactorings [4, 13, 27]. Studies recommend specific composite types to remove certain code smells [4, 15]. For example, Sousa et al. [26] recommend various *Move Methods* to fully remove a *God Class*. However, empirical studies report that developers often fall short in fully removing those code smells when they exclusively apply a sequence of *Move Methods* [4, 26]. When a recommended composite did not remove the target code smell, it affects the completeness of a composite. The composite completeness is a characteristic of a composite refactoring concerning the full removal of code smell(s) [3]. A complete composite is one that fully removes at least one code smell [3]. A composite is incomplete when the target code smell(s) remains [26].

The literature about composite (in)completeness is still scarce. First, there is no empirical evidence on how composite (in)completeness affects the internal structural quality. Developers need to know whether and what (in)complete composites can either effectively improve structural quality or adversely worsen it. Second, existing approaches have various limitations concerning the recommendations of effective complete composites. Their limitations include: (i) removal of a single code smell [4, 22, 26], (ii) the proposition of complete composites with a very limited number of refactoring types [4, 22, 26], and (iii) the recommendations are not driven by empirical knowledge on the software development practice [7, 18, 20, 25]. Thus, developers are reluctant to use existing complete composite recommendations [16, 23]. The aforementioned limitations have possibly misguided both developers and researchers on, respectively, applying composites and making solutions for composite refactoring practical. Based on that, this doctoral research focuses at empirically investigating on: (i) the positive and negative effects of composite (in)completeness on the internal structural quality [3, 6], and (ii) how to improve or even extend existing recommendations of complete composites.

On the effect of Composite (In)Completeness. We evaluated how 353 incomplete composites affect four internal quality attributes using ten code metrics [6]. We found the most incomplete composites (58%) tend to not change the internal quality attributes on smelly classes. This finding reveals that despite the incomplete composites not fully removing code smells, they at least maintain the structural internal quality. A relevant quantity (22%) of incomplete composite degrade internal quality attributes. We alert developers about what are incomplete composites that degrade each quality attribute. We collected 618 complete composites in 20 software projects [3]. We assessed whether the recommended complete composites [4, 26] introduced other code smells [3]. Our results present that about 36% of complete composites formed by *Extract Methods* introduced *Feature Envies* and *Intensive Couplings*. Unfortunately, the existing studies did not alert developers about this possible harmful introduction of complex code smells.

On the Complete Composite Recommendations. We found out that 64% of complete composites are formed by refactoring types that are not recommended previously [4, 26]. We then created a catalog with five recommendations of complete composites indicating (i) which code smells can be removed, (ii) what is the effect of each complete composite in the internal structural quality, (iii) refactoring types that are not recommended previously, and (iv) real examples of complete composites. As the next steps, we aim to improve and evaluate our catalog to help developers in practice.

2 MAIN CONCEPTS

Developers often then apply *composite refactorings or composite* [4, 11] to remove poor code structure, such as code smells [14, 30]. A composite is formed by refactorings that share at least a code element and are applied by the same developer [26].

Composite Completeness is a composite characteristic related to achieve the full removal of code smells [3]. **Complete Composites.** Recent studies have recommended composites to remove a single code smell type (the target code smell of the composite) [3]. For example, Sousa et al. recommend *Extract Method(s)* and *Move Method(s)* to remove Feature Envy [26]. We then consider that a composite is complete when it removes the target code smell(s). **Incomplete Composites.** A composite is incomplete when it fails to remove the target code smell(s) [6, 26].

Code modifications. Refactorings are often applied with other code modifications [21]. A code modification can be the addition of a method call, the creation of a variable. On the composite (in)completeness, it is interesting to know what are code modifications applied with (in)complete composites. This knowledge can help us to reveal whether, why, and how these code modifications are related to the composite (in)completeness. For Java projects, existing tools to collect code modification metrics use the library Eclipse's JDT 3.10.0 [12]. This library creates a parse Java source code into an Abstract Syntax Tree (AST) [24, 28].

3 PROBLEM STATEMENT

As aforementioned, a composite refactoring may be performed by developers with the goal of completely remove a code smell [4, 26]. Despite this goal, existing studies found that some smells remain after the application of composites, i.e., the completeness of composites to remove code smells is not achieved [4, 26]. This means that developers should be more cautious in avoiding to perform incomplete composites. If not avoided, the structure of the smelly program may get even worse. However, the literature is scarce on investigating the impact of complete composites in the overall internal software quality. For example, developers might assume that once a complete composite is performed, the improvement of the software quality is directly achieved.

Existing studies reported that single refactorings degrade the internal structural quality frequently [1, 9, 10]. As a composite refactoring usually is formed by several refactorings [8], this composite may degrade even more the structural quality of the program. Moreover, we also hypothesize that even a complete composite, which is focused in a particular smell, can introduce other types of smells, thereby also decreasing the internal software quality. However, there is a lack of in-depth studies about the completeness of smells. Existing studies not only ignore the influence of composite incompleteness, but also fail in revealing possible side effects of complete composites. This lack of empirical knowledge makes it difficult to assist developers with recommendations of how to successfully apply a composite without introducing any harm. This motivation leads us to the general research problem of this doctoral research: **Developers and researchers are misinformed or misguided about the (in)completeness of composite refactorings and their positive or negative effects**, detailed as follows.

Recent studies also reported that composites are frequently incomplete to remove code smells [4, 26]. For example, one of these studies reported that composites formed by multiple *Extract Methods* often fail to remove *Long Methods* [4]. As a consequence of these findings, previous work has provided a list of recommendations for complete composites [4, 22, 26]. Then, one should expect that by applying these recommendations the key internal quality attributes – such as coupling, cohesion, and complexity – should be improved or at least new code smells would not be introduced.

However, there is no empirical evidence about how (in)complete composites can affect the internal structural quality. As a consequence, both researchers and developers might not notice that even composites recommended by the literature are actually decreasing the internal quality of a program. If so, we should both revisit the effectiveness of existing complete composite recommendations and find proper ways of guiding developers on how to perform composite refactorings that effectively result in software with superior internal quality.

Research Problem 1: There is no empirical knowledge about the positive and negative effects of composite (in)completeness.

A few studies recommend composites to remove a single code smell [4, 22, 26], but their effectiveness has not been assessed. Previous pieces of work also recommend composites to remove multiple code smells simultaneously [7, 18, 20, 25]. However, these recommendations are automatically obtained without considering the knowledge or preferences of actual developers of a system. In other words, these recommendations are neither based in observed successful practices, not supported by proper empirical evidence. If these recommendations proposed in the literature are not effective, we are facing a misalignment between the theory and practice of composite refactoring.

This misalignment is already evident in recent studies reporting that developers are reluctant to use existing approaches to recommend complete composites [16, 23]. If empirical knowledge about the effectiveness of (in) complete composites is missing, researchers and practitioners will not be able to understand how to advance the state-of-the-art and state-of-the-practice. In particular, this lack of knowledge will make it difficult to properly recommend composites to fully remove multiple code smells according to the developers' needs. Without such support, developers will increasingly avoid to perform more complex refactorings in their projects.

Research Problem 2: Lack of recommendations to properly support developers on applying effective complete composites.

4 PROPOSED APPROACH

According to the research problems presented in Section 3, the study goal of this doctoral research is twofold: (i) investigate the effect of composite (in)completeness on internal structural quality, and (ii) recommend complete composites to remove multiple code smells. We then propose a catalog of empirically-driven composite recommendations to help developers to fully remove code smells.

4.1 Research Questions

We defined two research questions related to the research problems discussed in Section 3.

RQ_1. How does the composite (in)completeness affect the internal structural quality? The answer to this RQ aims to address the Research Problem 1. Studies present that single refactorings can negatively affect the internal structural quality, in particular, the degradation of internal quality attributes [1, 10] and the frequent introduction of additional smells [9]. A previous study also suggests that incomplete composites can affect internal quality attributes, such as code cohesion [4]. We investigate how incomplete composites affect internal quality attributes, and whether complete composites can introduce code smells while the target code smell is removed. With this investigation, we can have empirical evidence on how (in)complete composites affect the internal structural quality in practice. We can also understand whether and to what extent such composites introduce or not worse internal quality problems. As the main contribution, based on this obtained knowledge, we can guide developers and researchers on what and why the (in)complete composites can degrade or not the internal software quality.

RQ_2. How to recommend complete composites that better support developers in practice? The empirical knowledge on the effect of (in)complete composites is the basis to guide developers and researchers on composite recommendations (the answer of RQ_1). To have an understanding of how complete composites are applied can improve this guidance. By answering our RQ_2, using the empirical knowledge obtained in RQ_1, we can provide a better guidance on how to recommend complete composites, thus addressing Research Problem 2. Thus, we aim to aid developers by recommending complete composites using observations derived from the practice. We will create a catalog of complete composite recommendations and provide proper tool support for developers on applying their composites.

4.2 Proposal Steps

Figure 1 presents the steps of this dissertation proposal. The steps highlighted in green are steps already performed. The steps highlighted in orange are in progress. The steps highlighted in white will be performed. We detailed these steps as follows.

Step 1 - Database Construction. We performed two quantitative studies to create a database of composite refactorings, complete composites, and incomplete composites [3, 6]. For that, we collected data from 20 Java software projects according to the criteria detailed in [3]. We collected refactorings using Refactoring Miner 2.0 because this tool has a high accuracy [28]. We used the Organic tool to collect the code smells because this tool has a high accuracy [29]. We then performed two quantitative studies on (in)complete composites. For both studies, we created scripts to collect composites that are (in)complete to remove four smell types, namely *God Class*, *Complex Class*, *Long Method*, and *Feature Envy*. These smell types are the most common, and they can involve more than one class [9, 15]. According to previous studies [4, 26], a composite is classified as incomplete when it has at least one refactoring type recommended to remove the target smell type, but the smell was not removed. Similarly, our script detected a complete composite when the target smell was removed. In our first quantitative study, we collected 353 incomplete composites. We investigated the effect of incomplete composites on four internal quality attributes (cohesion, coupling, size, and complexity) using 10 code metrics based in [10]. In our second study, we collected 618 complete composites. They can remove the four smell types mentioned previously, but they can also remove the other 15 smell types detailed in [3]. We collected if these complete composites can also introduce these smell types while removing the target code smell. We also detected 18 refactorings types in complete composites that are not investigated previously [5, 22] to know if these refactoring types are often applied in practice. Our dataset is available on websites of our published studies [3, 6].

Step 2 - Catalog Creation. We created a catalog from our database, presenting the five most common types of complete composites applied in the practice. Currently, this catalog describes the following details: (i) the type of complete composite, (ii) the code smell to be removed, (iii) the code smells that can be introduced, (iv) the explaining about why these smells can be introduced, and (v) the description about to fully remove these smells. Our catalog is available on the website of our recent study [2, 3].

Step 3 - Catalog Improvement. We will add the five most common types of incomplete composites. We also plan to enrich the catalog with real examples for each type of (in)complete composites. At investigating on complete composite recommendations, we will collect code modifications (Section 2) that are applied with complete composites. It can help us to improve our complete composite recommendations. For example, we observed that complete composites formed by *Change Parameter Types* and *Extract Methods* removed *Long Method* and *Feature Envy*. These refactoring types were applied to reduce the method size because multiple parameters required many code lines. The developer applied the following modifications: removal of method calls and removal of variables' assignments that used these parameters. These code modifications were applied because the developer needs to remove these code smells. We will use existing tools to detect code modifications.

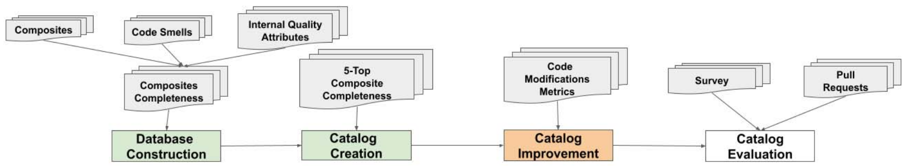

Figure 1: Dissertation Steps

Step 4 - Catalog Evaluation. We will apply survey and pull requests to evaluate our catalog. We aim to evaluate with developers that implemented smelly classes. We will select classes that are *God Classes*, methods that are *Long Methods* and have *Feature Envies* at the same time. A recent study presented that developers prefer to remove these code smells frequently [17]. **For Survey**, we aim to apply a questionnaire in software companies to evaluate how composites are applied in closed sources. We will then train developers on composite refactoring and code smells definitions. The smelly code will be presented to developers that will be asked to remove the code smells. Next, developers will answer a form about the refactorings applied. After, we will present and explain our catalog to developers. Other classes with the smell types will be presented to developers that should remove these code smells using our catalog and answer a questionnaire about the refactorings applied. **For Pull Requests**, we will submit them using our catalog on smelly code. We will evaluate how many pull requests will be accepted or rejected, and inquire the developers that assess these pull requests. Finally, we will evaluate if our catalog helps developers to remove code smells.

5 RESULTS ACHIEVED SO FAR

5.1 Effect of Composite (In)Completeness

Approach. On RQ_1, we performed two studies. In the first study [6], we found that 71% of composites were incomplete to remove *Feature Envies*, because they have at least one *Extract Method* without the application of *Move Methods*, as recommended previously by [4, 22, 26]. Most incomplete composites (58%) tend to not change the internal quality attributes on smelly classes. In a way, one could consider this fact problematic. However, this finding reveals that the incomplete nature of composites has possibly not harmed even further the program comprehensibility and other related quality attributes. On the remaining incomplete composites, we found out that 22% degrade internal quality attributes, 13% improve quality attributes, and 7% both degrade and improve them. Our results provide information to alert developers about the incomplete composites that can degrade internal software quality. In the second study [3], we found about 36% of complete composites formed by *Extract Methods* to remove *Long Methods* have introduced *Feature Envies* and *Intensive Couplings*. Unfortunately, the existing recommendations of composites did not alert developers about the possible introduction of other code smells. These results revealed instead of providing recommendations that ease the practice of applying composites to improve software quality, existing studies might be leading developers into making wrong decisions (code smell introduction).

Contributions. These results contributed to a better understanding of how composite (in)completeness affects internal structural quality. We provided (i) a dataset with 353 incomplete composites, and (ii) 618 complete composites that can remove target code smell(s), and introduce other code smells. Our studies provide a list of findings and guidelines to shed light on how to (i) extend existing refactoring tools for composite recommendations that can keep the internal quality attributes despite the not full removal of code smells, (ii) guide practitioners about when is not recommended to complete composites because it can increase the software degradation, and (iii) alert developers on the potential introduction of code smells while the target code smell is removed.

5.2 Completing Composites

Approach. On RQ_2, we performed one study [3]. We found that 64% of complete composites are formed by refactoring types that are not covered by previous studies [4, 26]. We also observed complete composites can remove multiple code smells. For example, composites with *Extract Method(s)*, *Move Method(s)*, *Change Return Type* (refactoring type not explored previously) removed *Feature Envies* and *Duplicated Code*. Our results present that existing composite recommendations are not aligned to the practice because (i) developers apply other refactoring types that are not recommended previously, and (ii) complete composites applied in practice can remove multiple code smells.

Contributions. We presented a catalog of common complete composites applied in practice. Our catalog shows detailed specifications on refactoring types that can be applied, code smells that can be removed, and code smells that can be introduced. We suggested that existing approaches recommend refactoring types that developers apply frequently, and composites that remove multiple code smells.

6 CONCLUSION AND NEXT STEPS

This doctoral dissertation proposes recommendations for composites to remove multiple code smells completely based on the practice. This doctoral research is currently in the sixth semester. We found: (i) incomplete composites can keep internal software quality, (ii) recommended complete composites can introduce code smells, and (iii) complete composites are formed by not recommended previously refactoring types in practice. We presented a catalog of composites to improve the internal software quality. We will evaluate our catalog from developers' perceptions using survey and pull requests.

Acknowledgments. I thank my advisor Alessandro Garcia and co-advisor Wesley K. G. Assunção, CNPq (141054/2019-0), CAPES (175956), and FAPERJ (22520-7/2016) for the support during this research.

REFERENCES

[1] Eman AlOmar, Mohamed Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In *13th ESEM (2019)*. 1–11.

[2] Ana Carla Bibiano. 2021. Complete Composite Website. https://anacarlagb.github.io/icsme2021-complete-composite/

[3] Ana Carla Bibiano, Wesley Assunçao, Daniel Coutinho, Kleber Santos, Vinıcius Soares, Rohit Gheyi, Alessandro Garcia, Baldoino Fonseca, Márcio Ribeiro, Daniel Oliveira, et al. 2021. Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects. In *37th ICSME*.

[4] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. A quantitative study on characteristics and effect of batch refactoring on code smells. In *13th ESEM (2019)*. 1–11.

[5] Ana Carla Bibiano and Alessandro Garcia. 2020. On the Characterization, Detection and Impact of Batch Refactoring in Practice. In *34th Brazilian Symposium on Software EngineeringSoftware Engineering - Doctoral and Master Theses Competition (SBES-CTD)* (Evento Online). SBC, Porto Alegre, RS, Brasil, 165–179. https://doi.org/10.5753/cbsoft_estendido.2020.14626

[6] Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, et al. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *28th ICPC*. 149–159.

[7] Glauber Botelho, Leonardo Bezerra, André Britto, and Leila Silva. 2018. A many-objective estimation distributed algorithm applied to search based software refactoring. In *CEC*. IEEE, 1–8.

[8] Aline Brito, Andre Hora, and Marco Tulio Valente. 2019. Refactoring graphs: Assessing refactoring over time. In *26th SANER (2019)*. 504–507.

[9] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *11th FSE (2017)*.

[10] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes? A multi-project study. In *31st SBES (2017)*. 74–83.

[11] Mel Cinnéide and Paddy Nixon. 2000. Composite refactorings for Java programs. In *14th ECOOP (2000)*. 129–135.

[12] Eclipse. 2021. Using the help system. https://help.eclipse.org/mars/index.jsp

[13] Eduardo Fernandes, Anderson Uchôa, Ana Carla Bibiano, and Alessandro Garcia. 2019. On the alternatives for composing batch refactoring. In *3rd IWoR, co-located: 41st ICSE (2019)*. 1–4.

[14] Francesca Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. 2015. On experimenting refactoring tools to remove code smells. In *16th XP, Scientific Workshops (2015)*. 1–7.

[15] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code* (1 ed.). Addison-Wesley Professional.

[16] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring: Challenges and benefits at Microsoft. *TSE (2014)* 40, 7 (2014), 633–649.

[17] Júlio Martins, Carla Bezerra, Anderson Uchôa, and Alessandro Garcia. 2021. How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective. In *35th SBES*. 54–63.

[18] Panita Meananeatra, Songsakdi Rongviriyapanish, and Taweesup Apiwattanapong. 2018. Refactoring Opportunity Identification Methodology for Removing Long Method Smells and Improving Code Analyzability. *IEICE TIS* 101, 7 (2018), 1766–1779.

[19] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *TSE (2004)* 30, 2 (2004), 126–139.

[20] Michael Mohan and Des Greer. 2019. Using a many-objective approach to investigate automated refactoring. *IST* 112 (2019), 83–101.

[21] Emerson Murphy-Hill, Chris Parnin, and Andrew Black. 2012. How we refactor, and how we know it. *TSE (2012)* 38, 1 (2012), 5–18.

[22] Willian Oizumi, Ana C Bibiano, Diego Cedrim, Anderson Oliveira, Leonardo Sousa, Alessandro Garcia, and Daniel Oliveira. 2020. Recommending Composite Refactorings for Smell Removal: Heuristics and Evaluation. In *34th SBES*. 72–81.

[23] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. 2019. Revisiting the refactoring mechanics. *Information and Software Technology* 110 (2019), 136–138.

[24] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *2010 IEEE ICSM*. IEEE, 1–10.

[25] Anshul Rani and Jitender Kumar Chhabra. 2017. Prioritization of smelly classes: A two phase approach (reducing refactoring efforts). In *3rd CICT*. IEEE, 1–6.

[26] Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana Carla Bibiano, Daniel Tenorio, Miryung Kim, and Anderson Oliveira. 2020. Characterizing and Identifying Composite Refactorings: Concepts, Heuristics and Patterns. In *17th MSR (2020)*.

[27] Daniel Tenorio, Ana Carla Bibiano, and Alessandro Garcia. 2019. On the customization of batch refactoring. In *3rd IWoR, co-alocated ICSE (2019)*. IEEE Press, 13–16.

[28] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *40th ICSE (2018)*. 483–494.

[29] Santiago A. Vidal, Willian Nalepa Oizumi, Alessandro Garcia, J. Andres Diaz-Pace, and Claudia Marcos. 2019. Ranking architecturally critical agglomerations of code smells. *Sci. Comput. Program. (2019)* 182 (2019), 64–85.

[30] Norihiro Yoshida, Tsubasa Saika, Eunjong Choi, Ali Ouni, and Katsuro Inoue. 2016. Revisiting the relationship between code smells and refactoring. In *24th ICPC (2016)*. 1–4.