On the Alternatives for Composing Batch Refactoring

Conference Paper · May 2019

DOI: 10.1109/IWoR.2019.00009

4 authors, including:

Anderson Uchôa
Pontifícia Universidade Católica do Rio de Janeiro
19 PUBLICATIONS **34** CITATIONS
SEE PROFILE

Ana Carla Bibiano
Pontifícia Universidade Católica do Rio de Janeiro
9 PUBLICATIONS **20** CITATIONS
SEE PROFILE

Alessandro Garcia
Pontifícia Universidade Católica do Rio de Janeiro
353 PUBLICATIONS **6,014** CITATIONS
SEE PROFILE

On the Alternatives for Composing Batch Refactoring

Eduardo Fernandes, Anderson Uchôa, Ana Carla Bibiano, Alessandro Garcia
OPUS Research Group, Informatics Department (DI), Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
Email: {emfernandes, auchoa, abibiano, afgarcia}@inf.puc-rio.br

***Abstract*—Code refactoring is often performed for improving code structures through code transformations. Many transformations, e.g., extracting or moving a method, are applied for at least partially removing code smells. Each code smell is a symptom of a poor code structure that makes hard to read and change the program. Developers often compose two or more interrelated transformations in conjunction (batch refactoring) rather than applying a single transformation. For instance, developers often compose method extractions with method motions to better organize the features realized by classes. We have recently observed cases of batch refactoring performed along with code review in open source projects. We then noticed that composing batches capable of fully removing code smells is quite challenging. Especially, it requires carefully discussing on how two or more transformations complement one another and what to expect from the batch effect on code smell. This position aims to reason about multiple alternatives to support developers on composing their batches. These alternatives should make it easier to compose batches that remove code smells. For this purpose, we exemplify the role of semi-automated tools in gradually recommending transformations, thereby guiding the batch composition in each alternative.**

***Keywords*-batch refactoring; code smell; code review.**

I. Introduction

Code refactoring is adopted by major companies aimed to improve code structures via code transformations [1] [2]. Many transformations are applied for at least partially removing code smells [3]. Code smells are symptoms of poor code structures that hinder reading and modifying the code [4]. Code smells may represent real maintenance threats but only 10% of single transformations can fully remove code smells [3]. Perhaps for this reason, developers often compose two or more interrelated transformations in conjunction, i.e., a *batch refactoring* (or batch) [2]. Notably, about 40-60% of code transformations are applied in batches [2]. An example of a batch is composing method extractions with method motions aimed to move an "envious" feature across classes [4].

Batch refactoring is often performed along with code reviews [5]. Shortly, code review is a practice aimed at detecting and removing problems in parts of a software project [6]. Through collaboration, developers discuss in rounds the refinements needed for removing problems, e.g., code smell instances [7]. After approval, the reviewed code is merged into the project's main source code [6]. We have found explicit mentions to poor code structures, especially code smells, along with code review discussions in the Google's Gerrit Code Review platform. In publicly available cases, reviewers suggest interrelated transformations (e.g., Extract Class, Move Method, and Inline Method composed) for removing a particular poor code structure [5]. Along discussions, developers debate about the suggested transformations. For instance, developers have to decide: apply, ignore, postpone, or ask for another suggestion? After rounds of discussions, a batch is finally composed.

The current batch refactoring support is scarce [3]. Tools like JDeodorant [8] recommend isolated code transformations, sometimes unrelated, one at a time. Thus, they make hard to reason about batches that fully remove smells. Only a few tools support the batch composition and they are quite limited [3]. FaultBuster recommends a full set of interrelated transformations [9]. However, the interaction with the tool is limited to accept or ignore the recommended batch. Refactoring Navigator [10] recommends transformations in rounds towards composing a batch. However, it supports major architectural refactoring rather than minor code transformations that are equally frequent in practice [3] [11].

This *position paper* aims to shed light on alternatives for composing batches in code review, because this context has many publicly available data for analysis. **Alternative 1:** we rethink the developer interactions with semi-automated tools in order to enable a customized batch composition. It works as an early code review aimed at shortening code review discussions. **Alternative 2:** we propose to support each reviewer in composing a particular batch so that all reviewers can decide the best one to apply. **Alternative 3:** we exploit the existing collaboration among developers in code reviews to promote discussions about the batch effect on code smells, guided by tool-recommended code transformations.

II. Motivating Example

Major and medium-sized companies adopt code review for coping with design and code problems [6] [12] [13], code smells included [5]. Two or more reviewers reason together about the quality of a program developed by another developer also known as the code owner [6]. The reviewers then forward all identified problems to the code owner for correction. Ideally, both code owner and reviewers engage in discussions about strategies for removing the detected problems [14]. Figure 1 illustrates a batch composed along with code review in the Couchbase project (Change 60201 available online[1] at Gerrit). This example shows how developers interact towards composing a batch for improving code structures along two consecutive commits (i and $i+1$). Here [5] is another example.

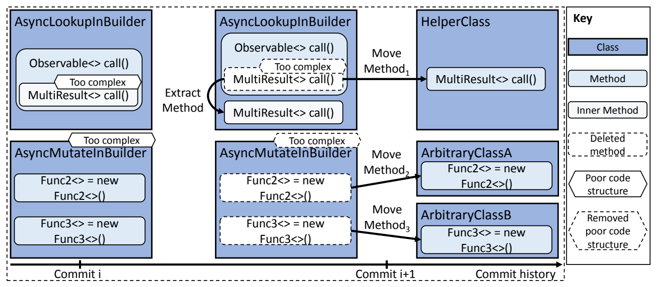


Figure 1. A Real Batch Example

Let us consider the *AsyncLookupInBuilder class. In Commit i, this class has an inner method named MultiResult<> call(), which is located into the Observable<> call() method. The inner method was found by Reviewer A as too complex to read and modify. For this reason, Reviewer A has recommended to move such method to a helper class (via the Move Method transformation), as illustrated in the quote below and extracted from Gerrit review comments. However, in Commit $i+1$, Reviewer B decided to postpone the recommendation and, instead, he extracted the inner method out of Observable<> call() in the same class.*

*That is a heavy callback – can we **move** that into a helper class?* – Reviewer A recommends transformation

*I'll **extract** it in its own constant, we can see later if it needs to be extracted in another class* – Reviewer B replies

Additionally in Commit i, let us consider the *AsyncMutateInBuilder* class. This class was implicitly considered as too complex to read and modify by Reviewer A. That is because of the methods encapsulated by this class, i.e. *Func2<> = new Func2<>()* and *Func2<> = new Func2<>()*. From the reviewer's perspective, each method should be moved to its own class. Thus, Reviewer A recommended the application of two transformations: Move Method$_2$ and Move Method$_3$. In this particular case, Reviewer B accepted both recommended transformations, which were performed in Commit $i+1$. These discussions among reviewers can be confirmed through the quotes below.

*I wonder if it makes it cleaner if those options down here are **moved** into separated classes each?* – Reviewer A recommends

Yeah, let's clean that up in another changeset if it becomes a problem – Reviewer B accepts recommendation

By the end of the code review discussions, the developers have composed and applied the batch $b = \{$Extract Method, Move Method$_2$, Move Method$_3\}$.

[1]http://review.couchbase.org/#/c/60201/

III. Challenges of Composing Batches

Challenge 1: Handling with Complex Batches. Batches may be complex to apply: 18% occur across project versions, 12% are composed of varied transformation types, and 50% have at least three transformations [3]. We assume that developer collaboration [14] is essential to compose batches, especially along with code review. Certain developers tend to concentrate the familiarity with specific parts of the code to remove smells [7]. Thus, our hypothesis is that knowledge exchange among many developers is mandatory to compose batches capable of (fully) removing smells. Unfortunately, the current support to batch refactoring [9] [10] limitedly supports such a knowledge exchange.

Challenge 2: Unknown Effect of Batches on Code Smells. Each single transformation is rarely capable to fully remove smells [3] [15]. Instead, it tends to either introduce (33%) or not fully remove (57%) smells. Especially along with code review, only a few developers may be unaware of the effect of transformations on the code structure. Although batches are frequent in practice [3], we know little about the batch effect on code smells. Such a limited knowledge may drastically affect the batch composition, especially to recommend transformations. For instance, one could wonder if the batch composed in Section II is a proper solution in that context. For instance, does the application order of transformations preserve the batch effect? We hypothesize that knowledge exchange could promote discussions on the code structure and how batches affect smells. A recommender that justifies the effect by isolated transformation could enhance existing batch composition tools [9] [10].

Challenge 3: Optimizing Batches is Tricky. Studies like [10] optimize batches by reducing certain attributes of batches, e.g., the number of code transformations into the batch. These studies aim to make batches shorter and easier to apply. A major threat to the existing optimization techniques is: they neglect the batch effect on code smells. Thus, our hypothesis is that current techniques may harm rather than improve code structures. "What attribute should we prioritize while optimizing batches?" and "Is there a trade-off between supporting the developer's motivation with refactoring and improving code structures via batches?" are questions that worth addressing via tooling support.

IV. Alternatives for the Batch Composition

Alternative 1: Developer vs. Tool (Early Code Review). *Problem* – Developers may spent too much effort with code review [6]. Some reviews just end up in blocking the reviewed code from being merged into the projects' main code [12]. Thus, reducing the code review effort while increasing effectiveness is desired. Alternative 1 aims to anticipate the smell detection and correction with the support of a semi-automated tool. The goal is reducing the code review effort in detecting and removing smells. An ideal tool should recommend transformations that gradually compose a batch. Alternative 1 acts during the code programming and, thus, the tooling support should constantly monitor code changes – e.g., via integrated development environment (IDE). Similar approach is successfully exploited by tools like [16] with other purposes than batch refactoring. Differently from existing batch composition tools [9] [10], the desired tool systematically promotes two key practices of code review [6] [14]: (1) the constant recommendation of code transformations and (2) the frequent discussion among developers about the recommended transformations.

Example – Figure 2(a) illustrates Alternative 1. The recommender constantly monitors smells during programming. In t_1, it was found a Large Class smell [4] instance affecting class C_1. The smell detection triggered the batch recommendation in successive interactions with the developer (i_1, i_3, and i_5). The developer accepted (i_2) and applied (t_2) a recommended Extract Class (via i_2), but he asked for a transformation other than Move Method recommended by i_3 (via i_4). He also accepted (in i_6) and applied (in t_3) Move Method as recommended by i_5. The resulting batch is $b_1 = \{$Extract Class, Move Method$\}$ that fully removes the Large Class code smell type.

Open Questions – What types of interaction (e.g., accept, reject, postpone) should be supported by recommendation? Studies like [10] could be a starting point for discussion. *What is the best time to recommend code transformations?* At each file save or before closing the software project? A previous work [16] guides the proposition of interactive tools aimed to avoid problems like information overload. However, practices for promoting interactions of developers are missing. *How to approach developers about smells affecting their produced code?* We could try to capture symptoms [13] of bad structural quality (e.g., smells) are valued in a project. For instance, we could reveal what kind of symptoms have been refactored out in previous versions of the project by using the association strategy defined in [11]. Then, we prioritize approaching developers with respect to those symptoms.

Alternative 2: Reviewer vs. Tool in Code Review. *Problem* – Code reviewers may have varied opinions on a given code structure. These opinions emerge from different knowledge degrees that developers have on the code, especially in terms of features realized by code element (e.g., a method or a class). Thus, each reviewer may compose a particular batch aimed at fully removing a code smell. As an example, the existing techniques for optimizing a batch mostly neglect the effect that different transformations orders, and even different batches, have on code smells. In this case, these techniques may not provide the best solution to developers aimed at improving code structures. Alternative 2 emerges with the purpose of letting each code reviewer to compose his batch, so that all developers can discuss the batch instance that best fits their needs.

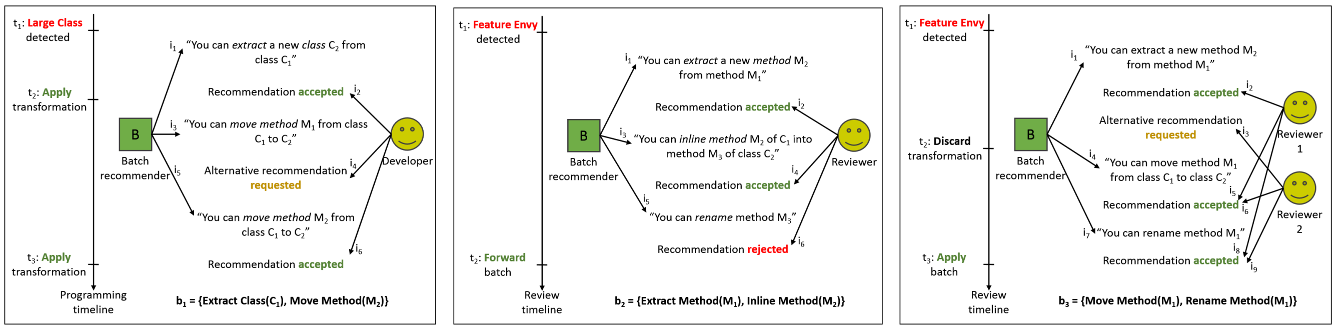
Example – Figure 2(b) illustrates Alternative 2. The batch recommender detects code smells in the reviewed code. In t_1, the recommender found a Feature Envy smell instance affecting method M_1. Feature Envy is (part of) a method that frequently uses features provided by other classes rather than the class that hosts the method itself [4]. The smell detection triggered the batch recommendation in successive interactions with the reviewer (i_1, i_3, and i_5). The developer accepted the Extract Method transformation recommended by i_1 (via i_2). He has also accepted Inline Method as recommended by i_3 (via i_4), but he rejected Rename Method recommended by i_5 (via i_6). Finally, the reviewer forwarded the batch to others in t_2. The resulting batch is $b_2 = \{$Extract Method, Inline Method$\}$ that fully removes Feature Envy.

Open Questions – How to measure the batch effect on code smells in such a way that developers are convinced to apply it? Strategies are desired to balance the developers' motivations with refactoring and the effect on code smells [3], which we plan to do in a recent thesis proposal [5]. *How to manage conflicts among reviewers that propose different batches?* In this case, the semi-automated tool could incorporate a voting system, similar to the one applied by Gerrit, so that developers can easily decide which batch should be applied at the end. Interactive search-based techniques like [10] can help to address both questions.

Alternative 3: Developer & Reviewers vs. Tool in Code Review. *Problem* – Many heads are sometimes better than a single one. In some code review tasks, e.g., detecting and correcting certain code smells, developers are more efficient when working collaboratively [7] [14]. Unfortunately, the current tools aimed to support the batch composition do not promote a collaborative work. Code review platforms like Gerrit allow developers and reviewers to discuss collaboratively. However, the collaborative batch composition has been unsupported by semi-automated tools integrated to these platforms. The unguided batch composition can increase the number of code review rounds, but also hinder the discussion about the batch effect on code smells. With Alternative 3, we explore the potential to enhance the batch composition through developer collaboration.

Example – Figure 2(c) illustrates Alternative 3 similarly to Alternative 2. In t_1, the recommender found Feature Envy affecting method M_1. The smell detection triggered the batch recommendation in successive interactions with all developers (i_1, i_4, and i_7). Although Reviewer 1 accepted the Extract Method recommended by i_1 (via i_2), Reviewer 2 asked for another recommendation (via i_3). This disagreement resulted in discarding Extract Method in t_2. After that, it was recommended Move Method in i_4, which Reviewers 1 and 2 accepted (via i_5 and i_6, respectively). It was recommended Rename Method in i_7 and both reviewers accepted it (via i_8 and i_9). The resulting batch $b_3 = \{$Move Method, Rename Method$\}$ was applied to fully remove Feature Envy.

Open Question – How to cope with many divergent reviewers' opinions about batches? Strategies for guiding


t1: Large Class detected
t2: Apply transformation
t3: Apply transformation
Programming timeline
B
Batch recommender
Developer
i1 "You can extract a new class C2 from class C1"
Recommendation accepted i2
i3 "You can move method M1 from class C1 to C2"
Alternative recommendation requested i4
i5 "You can move method M2 from class C1 to C2"
Recommendation accepted i6
b1 = {Extract Class(C1), Move Method(M2)}
t1: Feature Envy detected
t2: Forward batch
Review timeline
Reviewer
i1 "You can extract a new method M2 from method M1"
Recommendation accepted i2
i3 "You can inline method M2 of C1 into method M3 of class C2"
Recommendation accepted i4
i5 "You can rename method M3"
Recommendation rejected i6
b2 = {Extract Method(M1), Inline Method(M2)}
t1: Feature Envy detected
t2: Discard transformation
t3: Apply batch
Reviewer 1
Reviewer 2
i1 "You can extract a new method M2 from method M1"
Recommendation accepted i2
Alternative recommendation requested i3
i4 "You can move method M1 from class C1 to class C2"
i5 Recommendation accepted i6
i7 "You can rename method M1"
i8 Recommendation accepted i9
b3 = {Move Method(M1), Rename Method(M1)}

(a) Alternative 1 (b) Alternative 2 (c) Alternative 3

Figure 2. Some Alternatives for Composing Batches

the comparison of batches, especially in terms of their effect on smells, are desired. We highlight that b_3, which was composed through Alternative 3, differs a lot from b_2, composed via Alternative 2. In spite of that, both batches fully removed the same Feature Envy instance affecting M_1. We present this scenario on purpose to show how developers working alone and collaboratively may compose different batches. We then reinforce the need for guiding batch composition in many ways: from the isolated batch composition to the choice of the batch best fitting developers' motivations.

V. Final Remarks

This position paper illustrated three alternatives for composing batches aimed to remove code smells, especially in code review. We rely on our experience [5] with mining code review discussions available at Gerrit, besides the current knowledge on batches [1] [2] [3] for justifying the need for supporting batch refactoring in practice. We also provide insights on how to leverage the current batch refactoring support [9] [10]. We highlight that the exploited alternatives could be refined and adapted to remove other problems than code smells, such as design degradation [13].

Acknowledgment

We thank CNPq (grants 434969/2018-4, 312149/2016-6), CAPES (grant 175956), and FAPERJ (grant 22520-7/2016).

References

[1] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring: Challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, 2014.

[2] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, 2012.

[3] D. Cedrim, "Understanding and improving batch refactoring in software systems," Ph.D. dissertation, Informatics Department, PUC-Rio, Brazil, 2018.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Addison-Wesley Professional, 2018.

[5] E. Fernandes, "Stuck in the middle: Removing obstacles to new program features through batch refactoring," in *40th ICSE: Doctoral Symposium*, 2019, pp. 1–4.

[6] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *35th ICSE*, 2013, pp. 712–721.

[7] R. Oliveira, L. Sousa, R. de Mello, N. Valentim, A. Lopes, T. Conte, A. Garcia, E. Oliveira, and C. Lucena, "Collaborative identification of code smells: A multi-case study," in *39th ICSE: SEIP Track*, 2017, pp. 33–42.

[8] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of JDeodorant: Lessons learned from the hunt for smells," in *25th SANER*, 2018, pp. 4–14.

[9] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "FaultBuster: An automatic code smell refactoring toolset," in *15th SCAM*, 2015, pp. 253–258.

[10] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *24th FSE*, 2016, pp. 535–546.

[11] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *11th FSE*, 2017, pp. 465–475.

[12] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at Google," in *40th ICSE: SEIP Track*, 2018, pp. 181–190.

[13] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira, and C. Lucena, "Identifying design problems in the source code: A grounded theory," in *40th ICSE*, 2018, pp. 921–931.

[14] F. Zanaty, T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "An empirical study of design discussions in code review," in *12th ESEM*, 2018, pp. 11:1–11:10.

[15] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes? A multi-project study," in *31st SBES*, 2017, pp. 74–83.

[16] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *5th SOFTVIS*, 2010, pp. 5–14.