Evaluation of Move Method refactorings recommendation algorithms: are we doing it right?

Evgenii Novozhilov
Saint Petersburg State University
Russia
evgenii.novozhilov@gmail.com

Ivan Veselov
JetBrains Research
Higher School of Economics
Russia
idveselov@edu.hse.ru

Mikhail Pravilov
JetBrains Research
Higher School of Economics
Russia
mepravilov@edu.hse.ru

Timofey Bryksin
JetBrains Research
Saint Petersburg State University
Russia
t.bryksin@spbu.ru

***Abstract*—Previous studies introduced various techniques for detecting Move Method refactoring opportunities. However, different authors have different evaluations, which leads to the fact that results reported by different papers do not correlate with each other and it is almost impossible to understand which algorithm works better in practice. In this paper, we provide an overview of existing evaluation approaches for Move Method refactoring recommendation algorithms, as well as discuss their advantages and disadvantages. We propose a tool that can be used for generating large synthetic datasets suitable for both algorithms evaluation and building complex machine learning models for Move Method refactoring recommendation.**

***Index Terms*—automatic refactoring recommendation, move method refactoring, feature envy, algorithms evaluation, code smells, dataset generation**

I. Introduction

In software engineering, plenty of effort is put into minimizing human resources required to implement and maintain the required features. In object-oriented programming, a significant part of this effort concerns software architecture, which in time tends to drift from its original implementation and make the code less clear and more error-prone. To detect architectural degradation issues developers define so called code smells: specific code patterns or idioms that indicate possible architectural problems. One of the code smells is Feature Envy: a method suffers from the Feature Envy if it uses other classes more than its own class. One popular way to eliminate this code smell is to apply a refactoring that moves this method to a more appropriate class [6].

Automatic detection of code smells and refactoring opportunities seems like a well-researched field, a lot of papers that address this topic and propose various techniques have been published in the last two decades [2, 4, 15]. But software architecture is an emerging area of expertise that is highly subjective, which makes formal evaluation of refactoring recommendation algorithms a very difficult and controversial task. Since only a human developer is able to actually say whether this particular refactoring should be applied or not, most of the researchers choose expert assessment as a way to evaluate their work and compare it with other studies. Diverse opinions of various developers on what a good architecture is combined with different evaluation datasets lead to a very upsetting situation when different authors report results that simple don't correlate with each other. For example, in one of the recent studies [14] the authors examined papers on automatic refactoring recommendation published within the last ten years and found out that five different papers report five completely different evaluation results for the same tool (see Table I).

Paper	Precision	Recall
HIST [10]	0.65	0.71
JMove [12]	0.15	0.4
TACO [11]	0.57	0.69
c-JRefRec [17]	0.385	0.25
Domino [7]	0.76	n/a

TABLE I
JDeodorant's evaluation results in different papers [14]

In this paper we explore the reasons why such cases arise and discuss how the scientific community in this field could address this issue. Section II presents a comparative analysis of evaluation approaches used in research papers on Move Method refactoring recommendation. We come to a conclusion that a large and representative public dataset is an essential requirement for replicability and reproducibility of the obtained results. Section III examines already existing approaches to creating a dataset for Move Method refactorings recommendation algorithms, discusses requirements and storage format for such a dataset, and describes the proposed tool for injecting Feature Envy code smells into existing open source projects. We also provide a link to a dataset created using this tool.

II. Related work

A. Evaluation approaches

Evaluation is a crucial part of any research, which often involves tedious yet very important experimental work. Only an experiment can prove the value of the algorithm by comparing it to existing approaches. During the literature review, we have identified six major ways to evaluate refactoring recommendation algorithms.

1) Case studies on small projects where all refactorings are obvious: Researches manually explore some project and compile a complete list of existing code smells and possible refactoring opportunities. It could be an artificial example project showcasing selected code smells or a small real research project [15, 18]. This evaluation approach has obvious advantages: small projects are easy to comprehend, therefore labeling the methods does not require a lot of effort. But the results of the evaluation are not statistically significant since a small number of projects are studied (usually from 1 to 3) and these projects are highly specific, often created solely for the evaluation purpose. It could act as an example of how the proposed algorithm is designed to work but tells almost nothing about how it performs on complex real-world projects.

2) Expert assessment of the algorithm result on a real-world project: In these evaluations experts are asked to manually review the result of the algorithm for some existing real-world project and mark every obtained refactoring recommendation as valuable or not [15, 8, 1]. Since the list of recommendations is usually small, this approach also doesn't require a lot of human effort, but it's almost impossible to calculate the recall metric that way since experts only assess the algorithm's result, leaving all other possible refactoring opportunities unexplored.

3) Tracking software metrics: Another approach to evaluate algorithms is based on software metrics values [15, 2, 9, 1]. Most of them are cohesion and coupling metrics and their derivatives. The closer each metrics value gets to its ideal value after the recommended refactoring has been applied, the more valuable this particular refactoring is considered to be. This evaluation approach could easily be automated, but it depends highly on the selected metrics set, which could be very subjective. Moreover, this kind of evaluation does not allow to measure precision, recall or any other statistical metrics since most often it is run on an unlabeled dataset.

4) Evaluation on refactorings mined from historical data: The idea of this evaluation approach is to collect refactorings that were actually used by developers in the past in real projects [7]. To gather them, each modification in the version control system is analyzed and two consecutive snapshots S_1 and S_2 are compared to see if a particular refactoring took place in this modification. Then the algorithm is run on the S_1 snapshot to see if this refactoring will be recommended or not. Based on existing refactoring mining tools (e.g., RefactoringMiner [16]), this evaluation method doesn't require much human effort, but for a whole project (which could be pretty large) only one or several refactorings are expected to be found, which is not very effective. In addition, we also fail to accurately measure precision, recall and other metrics.

5) Evaluation on a labeled dataset: This evaluation approach is the most widely used one [1, 4, 5, 10]. A group of experts (often Master's or PhD students or industry experts) is asked to go through some large open source projects and mark each method, depending on whether it should be moved away or not. This activity results in a fully labeled dataset, which can be used to calculate metrics like precision or recall. But the obvious drawback here is that is takes a huge amount of effort to label large number of real-world projects this way, especially if one wants to have unbiased labels, meaning that each project should be labeled by more than one developer expert.

6) Evaluation on a dataset with artificially introduced code smells: This approach is based on the assumption that the proportion of methods with Feature Envy to all methods in a real-world long-term project is rather small, and if a method will be moved from one class to another, a Feature Envy code smell will most likely be introduced. The algorithm is evaluated by how many of the moved methods it recommends to move back to their original class [12, 8, 17]. The advantage of this approach is that the dataset could be gathered automatically, and it could be as large as the evaluation needs. But this approach obviously follows the assumption that all methods are initially positioned correctly, which is never true in large real-world projects. Besides, when a method is moved randomly, you can make a case of Feature Envy code smell too obvious, which makes the evaluation too artificial.

As the literature review shows, almost all papers inevitably discuss how to build a dataset in their evaluation. Most of the authors collect their own data, which leads to different evaluation results. One possible solution to this problem is to create an open dataset or a tool that allows to create it automatically. With such labeled data publicly available, the evaluation of any new algorithm would require significantly less effort, and would finally make comparison of different approaches accurate, which could lead to a better adoption of these approaches by the industry.

III. Dataset generation

A. Dataset generation approaches

As was mentioned before, one way to create the described dataset is to manually label each code smell occurrences in each project. Several authors used this approach in their evaluation [11, 4]. Unfortunately, these datasets are rather small: for each code smell type they usually have roughly couple of hundreds entries. Evaluation on such datasets simply can't be representative enough to capture all the peculiar properties of the compared algorithms. Besides, machine learning approaches, which have been used for code smells detection more often recently, usually require much more data for training.

Another promising possibility is to create a dataset of Move Method refactorings performed in real-world projects in the past. To achieve this, we fetched 15 top-rated GitHub open

source Java projects and applied RefactoringMiner tool to get possible refactorings from these projects' histories (23447 commits were processed). Unfortunately, despite reporting the best results for such mining tools, RefactoringMiner provided a lot of false-positives while detecting Move Method refactorings (for instance, when a method was moved to a new class, which is a clear case of Extract Class refactoring, and several others). After thorough filtering, less than 20 correct items were left out of 1348 initially detected Move Method refactorings. This is significantly less than the size of manually constructed datasets, which suggests automatic dataset creation from commit history is not very efficient with the tools that are currently available.

The third way to create a dataset is to generate synthetic code smells based on real-world projects. This approach was used in several studies (for example, [8, 13]) and can potentially be scaled. As mentioned in Section II-A6, the idea is to displace methods from their containing classes to some other classes. Surely, this leaves us with an artificial dataset, but it seems like the only feasible way that allows to collect large amount of data and that does not require tremendous human effort. Moreover, as long as the selected projects have good architecture and most of their methods are contained in appropriate classes, the noise in the resulting dataset should not be significant.

In this work we have chosen the last approach as the only viable possibility to generate large amount of data automatically.

B. Dataset structure

Different approaches require different information extracted from the input data. The vast majority of algorithms use numerical features calculated for each point in the dataset (fields, methods, classes, etc.). These features and data points vary from one algorithm to another. For example, JDeodorant [15] uses Jaccard distance to measure similarity between a method and a class. The same formula was reused in [8] to calculate input features for a classifier based on a deep neural network. On the other hand, in [4] the authors use standard object-oriented metrics as inputs to statistical machine learning models. Methods that are based on textual information [2, 11] require source code of the entities to statically analyze them. Clustering approaches [3] might require the whole project to perform their analysis. Also, the entire project history might be used to evaluate history-based algorithms [7].

As long as the data is synthetic and is supposed to suit as much existing approaches as possible, the most generalized and compact way to store it is to store only the information that is needed to generate all data points on demand. A special instance of the dataset could be created for each particular approach. For example, a set of independent method relocations could be applied to a project to generate a code base with a particular number of code smells in it. Then this modified project could be used to evaluate some clustering-based approach. Or a special instance of the dataset could be created for a metrics-based approach, consisting of vectors of metrics values for each class or method. Unfortunately, it is quite hard to generate a reasonable dataset for history-based approaches, so we leave them out of scope in this work.

C. The proposed tool

To support our idea we have implemented a tool[1] that accepts a Java project as input and produces all possible method relocations as output. More precisely, the tool returns a list of all methods that can be moved from their class to some other class in the project. Each entry in this list is a value containing a method, its original class, and a nonzero number of other classes this method can be placed to.

Not every method can be moved to some other class and is suitable for the output dataset. First of all, static methods are not considered as candidates since they represent global functions and differ from the usual instance methods. The most usual destination (target) class candidates for a non-static method are classes of this method's parameters: it's usually a simple technical task to move a method to such a class, as objects of these classes are definitely present in the context of every call of the moved method. So, if a methods doesn't have any parameters it is also not considered as a candidate.

Even if a method has a target class, it may not be possible to automatically move this method out of its original class. For example, a method may access a private field of its current class. In this case, moving this method away will produce code that can't be compiled. One way to fix this would be to generate additional getter methods for the used private fields, but we believe that this kind of modifications change the code structure too much comparing to the original project. These methods are also considered non-movable.

Another example of a non-movable method is a class constructor. It is quite obvious that a constructor of one class can't be moved to another class. Abstract methods, getters, setters, and overridden methods are also considered non-movable and are filtered out from the candidate list.

There are also methods that are movable and have valid target classes, but there is no practical value to add them to the dataset. For example, a method's body may consist of a single statement that throws an exception; such methods are also filtered out. It is very unlikely that any algorithm could determine which class is the right one for such a method looking only at its code. Empty and delegation methods are removed from the list for the same reason.

Moreover, the tool ignores all methods from annotations, builders and empty classes. Test classes are ignored as well.

Table II presents the results of the tool run on several open source projects from GitHub. The numbers of detected movable methods could seem small compared to the whole number of methods. However, the whole process is fully automated, which means that this data could be easily scaled by simply adding new repositories.

To construct such a dataset from the MoveMethodGenerator's output we have used the IntelliJ Platform SDK to

[1]A tool for generation of Move Method refactorings recommendation datasets, https://github.com/ml-in-programming/MoveMethodGenerator

automate the movements of the methods. The obtained dataset is also available online[2].

Project	Commit	Methods in project	Movable methods
Apache Cayenne	a4c6d99	16904	140
JUnit	91e8cd6	2821	25
PMD	db2348b	9768	133
Spring	33cbe2e	28941	198

TABLE II
TOOL OUTPUT FOR SEVERAL OPEN SOURCE PROJECTS

IV. CONCLUSION

As can be seen from the literature review, the refactoring recommendation field lacks a commonly used technique to compare research results. This makes it hard to understand the actual state of research and integrate state-of-the-art refactoring recommendation approaches into modern software development tools. In this paper we provide an overview of several evaluation techniques being used in existing papers. Based on their advantages and disadvantages we propose a tool capable of generating synthetic datasets from real-world projects that could be used both for the evaluation of the Move Method refactoring recommendation approaches and for new research based on machine learning techniques that require large amounts of data (e.g. deep learning models).

REFERENCES

[1] Vahid Alizadeh, Marouane Kessentini, Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 2018.

[2] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, 2014.

[3] Timofey Bryksin, Evgenii Novozhilov, and Aleksei Shpilman. Automatic recommendation of move method refactorings using clustering ensembles. In *Proceedings of the 2Nd International Workshop on Refactoring*, IWoR 2018, pages 42–45, New York, NY, USA, 2018. ACM.

[4] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[5] Francesca Arcelli Fontana and Marco Zanoni. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 128:43–58, 2017.

[6] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[2]The dataset obtained via the MoveMethodGenerator tool: https://github.com/ml-in-programming/MoveMethodDataset

[7] Hui Liu, Yuting Wu, Wenmei Liu, Qiurong Liu, and Chao Li. Domino effect: Move more methods once a method is moved. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 1–12. IEEE, 2016.

[8] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 385–396. ACM, 2018.

[9] Mark OKeeffe and Mel O Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[10] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.

[11] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[12] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. Recommending move method refactorings using dependency sets. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 232–241. IEEE, 2013.

[13] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. Jmove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, 138:19–36, 2018.

[14] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Ten years of jdeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14. IEEE, 2018.

[15] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[16] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM.

[17] Naoya Ujihara, Ali Ouni, Takashi Ishio, and Katsuro Inoue. c-jrefrec: Change-based identification of move method refactoring opportunities. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 482–486. IEEE, 2017.

[18] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3):501–532, 2016.