Automatic Recommendation of *Move Method* Refactorings using Clustering Ensembles

Timofey Bryksin
JetBrains Research
Russia
t.bryksin@spbu.ru

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

Aleksei Shpilman
JetBrains Research
Russia
alexey@shpilman.com

ABSTRACT

In this paper, we are approaching the problem of automatic refactoring recommendation for object-oriented systems. An approach based on clustering ensembles is proposed, several heuristics to existing algorithms and to filtering and combining their results are discussed. Experimental validation of the proposed approach on an open source project is presented. The obtained preliminary results illustrate that the introduced approach could be successfully used to improve existing integrated development environments, providing developers with one more tool to reduce the complexity of their projects. The paper concludes with a discussion on the applicability of such automatic refactoring recommendation approaches to real-world software developed using common object-oriented techniques.

CCS CONCEPTS

• **Software and its engineering → Object oriented development**; *Object oriented architectures*; *Software maintenance tools*;

KEYWORDS

automatic refactoring recommendation, clustering ensembles, object-oriented architecture

ACM Reference Format:
Timofey Bryksin, Evgenii Novozhilov, and Aleksei Shpilman. 2018. Automatic Recommendation of Move Method Refactorings using Clustering Ensembles. In *Proceedings of the 2nd International Workshop on Refactoring (IwoR '18), September 4, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3242163.3242171

1 INTRODUCTION

Refactoring is a well-known tool helping to improve code readability and reduce its internal complexity, which makes systems easier to extend and maintain. A number of refactoring techniques are already known, and modern integrated development environments (IDEs) provide support to perform such technical tasks as moving a method or extracting a class automatically.

In this research, we are trying to make one more step in helping software developers make their code less complicated and tangled by analyzing the code of a project opened in an IDE and suggesting developers possible directions of refactoring. But there still is no commonly accepted criteria or a formalism describing "good object-oriented code". Dozens of metrics were created measuring data encapsulation, abstraction, cohesion, coupling and other intrinsic key characteristics of code. Some of these metrics are built solely on opinions on what good architecture is ([16] even shows that refactoring metrics correlate weakly with human judgment), a programmer can usually find a set of metrics that represent his or her view on this matter.

The objective of this paper is to use clustering ensembles for automatic identification of *Move Method* refactorings for object-oriented projects. This allows to mitigate the problem of using just one single quality metric or one single refactoring detection approach and to combine their strengths together. Based on this approach a plugin for IntelliJ IDEA has been implemented that runs a number of clustering algorithms, generates a list of possible refactorings, ranks them according to an introduced metric of accuracy and presents them to a developer using IDEA's tool windows interface. If the developer finds one or more of suggested refactorings suitable for his or her needs, they can be selected and applied automatically (if this refactoring can be applied using IDEA's internal refactoring tools).

The remainder of the paper is organized as follows. Section 2 provides brief overview of automatic refactoring recommendation field, mentions existing approaches and tools. Section 3 describes the suggested approach: reviews algorithms selected for the ensemble and explores introduced heuristics and modifications to discussed algorithms. Section 4 provides an evaluation of the implemented ensemble on an open-source project. Section 5 discusses the evaluation results and applicability of such an approach for the use in enterprise IDEs. Section 6 summarizes our work and outlines possible directions for future work.

2 RELATED WORK

Recommendation of refactoring opportunities in software projects is a well-researched domain. Several techniques to detect design defects and suggest appropriate refactorings have been presented: Bayesian belief networks [7], game theory [3], multi-objective genetic programming [8], clustering [11, 15] and a number of different search methods (e.g., [1, 13]). The most widely researched refactorings are *Move Method* [5, 17], *Extract Method* [4, 12] and *Extract Class* [3, 15]. Several papers present similar ideas to package-level refactorings [2, 14]. However, implementations are not usually publicly available for most of published papers, thus it is basically not possible to provide proper assessment of multiple algorithms together.

Several tools exist that detect "code smells" and architectural defects like PMD[1], iPlasma [2] or JDeodorant [3]. PMD is a highly customizable static analyzer, so it can detect problems, but does not provide any solutions. iPlasma is a stand-alone application, hence is not easily integrated into development process. JDeodorant is a plugin for Eclipse IDE, and it is able to suggest *Extract Method*, *Move Method* and several other refactorings.

Like JDeodorant our solution is a plugin for an industrial IDE, which makes it easy to use for software developers, and using ensembles of different algorithms we can generate a more diverse set of results.

3 OUR APPROACH

3.1 Selected Algorithms

During literature review we have selected three algorithms that reported the highest ratio of justifiable refactorings: ARI [10], HAC [9], and CCDA [14].

The ARI (Automatic Refactoring Identification) algorithm introduces a vector space model. Each method and class in a system is represented by a vector, which elements are values of different software metrics: Relevant Properties (RP), Depth in Inheritance Tree, Number of Children, Fan-In and Fan-Out [11]. RP for a method is a set that consists of the method itself, the class where the method is defined, all fields and methods accessed by this method, and all methods that override this method. RP for a class is a set that consists of the class itself, all fields and methods from this class, all interfaces implemented and all classes extended by this class. So an entity s_i is represented by a vector $(s_{i1}, s_{i2}, s_{i3}, s_{i4}, s_{i5})$, where s_{i1} is a Relevant Properties set and other components are metrics values in some particular order. A semi-metric function shown in Formula 1 measures dissimilarity between two entities from the system.

$$d(s_i, s_j) = \begin{cases} 0, & \text{if } i = j \\ \sqrt{\frac{1}{m} \cdot \left(1 - \frac{|s_{i1} \cap s_{j1}|}{|s_{i1} \cup s_{j1}|} + \sum_{k=2}^{m} (s_{ik} - s_{jk})^2\right)}, & \text{if } s_{i1} \cap s_{j1} \neq \emptyset \\ \infty, & \text{otherwise} \end{cases} \quad (1)$$

The algorithm initially places each class into a separate cluster and then tries to put each method into the closest cluster according to Formula 1 or to put it into a new cluster if the distance to all existing clusters is greater than 1. Finally, the algorithm searches for classes whose distance between each other is less than 1 and merges them. The resulting partition is compared with original code structure and three types of refactoring are identified: *Move Method*, *Extract Class* and *Inline Class*.

The HAC (Hierarchical Agglomerative Clustering) algorithm uses the same vector space model and distance function as ARI, but calculates the distance between two clusters using *complete link* linkage method. Initially, each program entity (a method or a class) is placed into a separate cluster. It calculates distances between all pairs of clusters, takes the minimum distance and merges these clusters if the distance is less than 1 and skips if otherwise. This step is performed as long as a pair of clusters with the distance less than 1 exists. The rest is similar to ARI: compare the resulting partition with the original architecture and identify the same three types of refactoring.

The CCDA (Constrained Community Detection Algorithm) was originally used for refactoring identification at the package level [14]. It employs the formalism of software networks to represent classes and dependencies between them: each class is represented by a node in the network and an edge between nodes is created if there is a dependency between the corresponding classes. Edges are weighted to specify the strength of the dependency. The algorithm tries to detect communities within such a network: sets of nodes that have a higher density of edges within each set than between them. Initially, each class is placed into a separate community, and then the algorithm starts to move classes between communities to optimize a quality index for the given partition. In CCDA, $Q = \sum_i (we_{ii} - wa_i^2)$ is used as quality index, where we_{ii} is the fraction of the total weight of edges that connect two nodes within community i, and wa_i is the fraction of the total weight of edges that have at least one endpoint within community i. In our research CCDA was adopted to work at the class level, redistributing methods between classes instead of redistributing classes between packages.

Ultimately all selected algorithms are trying to enforce well-known "high cohesion, low coupling" object-oriented rule: via refactoring suggestions guide the project to a state where there are fewer dependencies between objects and method interaction is mostly performed internally. CCDA achieves this by creating clusters that are as dense as possible, ARI and HAC are trying to form clusters of "similar" methods using Fan-In, Fan-Out and Relevant Properties metrics values.

3.2 Implementation Details

Despite ARI and HAC being able to detect several types of refactoring we decided to currently limit ourselves only by *Move Method* refactorings. First of all, this type of refactoring is a common ground for all selected algorithms, and secondly, we did not want to overcomplicate the resulting tool: detection of a different type of refactorings requires different user interactions and could confuse our users. So we decided to start with what every developer understands well: moving methods between classes.

Several novel heuristics were introduced to ARI and HAC based on experimental runs to achieve more relevant results. First of all, weights were added to different types of components in the Relevant Properties metric:

- public static fields and methods are not added to other entities' Relevant Properties sets. Dependencies on such entities are not considered strong enough to justify moving a method somewhere else;
- weight value 1 is assigned to getter methods;
- weight value 2 is assigned to private static methods;
- weight value 4 is assigned to public non-static methods;
- weight value 6 is assigned to non-public non-static fields and methods and to the entity itself when it is added to its own Relevant Properties.

[1]PMD static code analyzer: https://pmd.github.io/
[2]iPlasma code analyzer: http://loose.upt.ro/reengineering/research/iplasma
[3]JDeodorant Eclipse plugin: https://marketplace.eclipse.org/content/jdeodorant

Calculation of cardinality for Relevant Properties' intersection and union was also redefined accordingly to represent not only the number of elements in the set but also their weights. If an entity belongs to an intersection of RP_i and RP_j, it can have different weights in each of these sets. For the intersection, a minimum of these two weight values is selected and the overall intersection cardinality value equals to the sum of its entities' weights. The cardinality of the union of two Relevant Properties sets equals to the sum of weights of non-overlapping entities plus the cardinality of the intersection of these sets.

This modification allows assessing the strength of dependencies between entities. Taking them into account allows to filter out refactorings that are unjustifiable.

The second major heuristic introduced in this research is the *accuracy* value of suggested refactorings — a metric showing how relevant each suggested refactoring is believed to be according to the algorithm (or algorithms) that suggested it. It is calculated as follows.

- For ARI and HAC an auxiliary *difference* metric is calculated as a difference between distances to first and second closest clusters. If an entity is decided to be moved into some cluster, the accuracy value of this refactoring is set to the minimum of $5 * difference$ and 1.
- For CDDA a density value is calculated for each cluster: the maximum number of entities from the same class is divided by a number of overall entities in this cluster. If an entity is decided to be moved into some cluster, this cluster's density value becomes the accuracy value of this refactoring.

These formulas were empirically selected as they provided the best results among various experimental runs of the algorithms.

The third heuristic concerns representation of results. At this point, we merge all individual algorithms into a single ensemble which provides the final result. Each algorithm produces a set of refactorings in the form of $(entity, target, accuracy)$ triplets. All sets are merged, grouped by the *entity* value and each accuracy value is squared. If a particular refactoring (an $entity\text{–}target$ pair) was suggested by more than one algorithm, their *accuracy* values are summed up, divided by a number of algorithms suggested this refactoring and the square root is calculated. Each refactoring's accuracy value is calculated as follows:

$$accuracy = \sqrt{\frac{\sum_{i=1}^{n} accuracy_i^2}{n}}, \tag{2}$$

where n is a number of algorithms suggested this refactoring and $accuracy_i$ is an accuracy value provided by the i-th algorithm. Then for each entity, the refactoring with higher accuracy value is selected as the most relevant one. If this value is more than an empirically selected threshold of 0.5, then the refactoring is considered worthy.

3.3 Implemented IntelliJ IDEA Plugin

All three algorithms were implemented as a plugin for IntelliJ IDEA[4]. The source code is available on GitHub[5]. Our plugin allows to run the algorithms on the project code currently opened in the IDE (currently only Java projects are supported), shows suggested refactorings in a tool window, allows to navigate to corresponding parts of the project and, if asked, tries to apply selected refactorings automatically using IDEA's existing refactoring tools.

[4]https://plugins.jetbrains.com/plugin/10411-architecturereloaded
[5]https://github.com/ml-in-programming/ArchitectureReloaded

4 EXPERIMENTAL EVALUATION

Papers on automatic refactoring recommendation employ two major approaches to evaluate proposed algorithms. Within the first approach, a number of artificial code fragments are created stating obvious contexts where a particular refactoring should be applied. Running an algorithm on these fragments confirms that it can handle such cases well. To our best knowledge there is still no dataset of such nature publicly available, so the more examples researchers come up with, the better their evaluation will be. However, this kind of evaluation still tells very little about how an algorithm behaves on large real-world projects (i.e., its target domain).

The second approach involves running algorithms on existing projects, usually well-known and open-source. But, as mentioned before, there is no common definition of a good architecture or good code, so there is no sure way to tell if each particular refactoring is "good" or "bad". This fact does not allow us to use well-known statistical and machine learning metrics like precision, recall, F1 score, etc. since we simply can estimate neither how many worthy refactorings in this particular system can be recommended nor how worthy are refactorings recommended by a particular algorithm. The only way to evaluate the results of this approach is to ask expert developers to rate the result and to be as objective as possible, arguing their decisions.

The solution proposed in this paper was also tested on a number of real-world projects. One of them is jackson-core[6] Java library which is a part of composite Jackson Data Processor project used for JSON parsing and generating. Jackson project is known as a "best JSON parser for Java" and used in multiple projects all over the world. Library source code was picked from its public repository hosted on GitHub matching revision eb86a5dc. It consists of 86 classes and 1767 methods.

For jackson-core we received 6 refactorings with accuracy values higher than 0.5, 5 of which in our opinion are justifiable:

- Moving JsonGenerator.writeTypeSuffix() to WritableTypeId or even extracting part of it is reasonable because this method performs a large number of operations with its argument of WritableTypeId type and potentially breaks the concept of incapsulation.
- Moving _loadMoreGuaranteed() which is a member of both UTF8StreamJsonParser and ReaderBasedJsonParser to its grandparent class ParserMinimalBase is reasonable to address code-duplication issue since both methods are identical.
- Movement of Base64Variants.valueOf() to Base64Variant class is feasible because the mentioned method basically constructs Base64Variant object based on the input string. This particular method can be implemented as a constructor for Base64Variant or a factory method.
- Moving JsonParserSequence.switchAndReturnNext() to it's base class JsonParserDelegate if feasible because this method

[6]Jackson-core Java library: https://github.com/FasterXML/jackson-core.

almost completely operates with JsonParserDelegate object and returns a result based on its public methods.

For jackson-core project each individual algorithm's results are as follows:

- in top 5 ARI results only 3 items are present in the ensemble's result, and only 2 of them are feasible;
- in top 5 HAC results 2 refactorings are also present in the ensemble's result, and both of them are feasible;
- in top 5 CCDA results nothing presents in the ensemble's result. However there is a JsonGenerator.writeTypeSuffix() in top 11.

As we can see, the ensemble manages to combine the best suggestions from individual algorithms which almost have no items in their top 5 results' intersection and present more balanced result.

5 DISCUSSION

Automatic refactoring identification is complicated by the fact that there is no sure way to assess the result automatically and only a human developer can properly judge whether this particular refactoring should be applied or not (and different developers might judge differently).

In a preliminary user evaluation of our refactoring plugin by a dozen professional software developers, we received very positive feedback concerning the results filtering feature. As mentioned before, only refactorings that got total accuracy value higher than 0.5 were initially shown (if there were no such refactorings, the top 5 were shown regardless of their accuracy values), but there is a slider allowing to set any accuracy threshold value. It gave developers an opportunity to consider most "popular" suggestions first and review all others later if they were interested.

Some of the refactorings that were suggested during the evaluation could not have been applied as they were (for instance, moved methods referenced private fields or called other methods that were not possible to move), but still might give developers a hint on how to improve their code.

According to the nature of selected algorithms (as mentioned in Section 3.1) the ensemble aggressively attacks methods that extensively interact with objects passed as their arguments or created within the method. While in most cases that is a valid suggestion, several design patterns like Builder or Visitor also fall into this category (see [6] for more such examples). For example, all a Builder object does is construct an object, set its values and return the object. So from the algorithms' point of view that is breaking the encapsulation. Most of such code currently is suggested to be moved into this object's class, turning this method into some Factory Method design pattern.

6 CONCLUSION AND FUTURE WORK

Software refactoring can significantly improve quality of code. For the last decade there has been a lot of research trying to suggest refactorings automatically, many algorithms were proposed employing different techniques, but few of them found their way to enterprise tools that software developers use in their everyday work. In this paper we propose a way to combine existing knowledge in this field, creating ensembles of several algorithms. This approach has both academical and practical value: implemented tool can be used to compare different refactoring identification algorithms and to apply them to existing projects.

Although our approach showed promising results, there is room for further improvements. Future work will include considering different algorithms, identifying and exploiting their individual strengths to build a better ensemble, learning developer's refactoring preferences and adapting suggestions accordingly, and detection of other refactoring types supported by IntelliJ IDEA's SDK, for example, *Move Field*, *Extract Method* or *Pull Up/Push Down Method*.

REFERENCES

[1] 2008. Search-based Refactoring for Software Maintenance. *Journal of Systems and Software* 81, 4 (April 2008), 502–516.

[2] Abdulaziz Alkhalid, Mohammad Alshayeb, and Soliman A. Mahmoud. 2011. Software refactoring at the package level using clustering techniques. *IET Software* 5 (June 2011), 274–286(12). Issue 3.

[3] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y. G. GuÃľhÃľneuc. 2010. Playing with refactoring: Identifying extract class opportunities through game theory. In *2010 IEEE International Conference on Software Maintenance.* 1–5. https://doi.org/10.1109/ICSM.2010.5609739

[4] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou. 2017. Identifying Extract Method Refactoring Opportunities Based on Functional Relevance. *IEEE Transactions on Software Engineering* 43, 10 (Oct 2017), 954–974. https://doi.org/10.1109/TSE.2016.2645572

[5] Christian Marlon Souza Couto, Henrique Rocha, and Ricardo Terra. 2017. Quality-oriented Move Method Refactoring. In *BENEVOL 2017, 16th BElgian-NEtherlands software eVOLution symposium.*

[6] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. 2016. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 609–613.

[7] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In *2009 Ninth International Conference on Quality Software.* 305–314. https://doi.org/10.1109/QSIC.2009.47

[8] Usman Mansoor, Marouane Kessentini, Bruce R. Maxim, and Kalyanmoy Deb. 2017. Multi-objective Code-smells Detection Using Good and Bad Design Examples. *Software Quality Journal* 25, 2 (June 2017), 529–552.

[9] Zsuszanna Marian. 2012. A study on hierarchical clustering based software restructuring. *Studia Universitatis Babes-Bolyai, Informatica* 57, 2 (2012).

[10] Zsuzsanna Marian, Gabriela Czibula, and Istvan Gergely Czibula. 2012. Using software metrics for automatic software design improvement. *Studies in Informatics and Control* 21, 3 (2012), 250.

[11] Z.-E. Marian. 2014. *Machine learning based software development.* Ph.D. Dissertation. Faculty of Mathematics and Computer Science, Babes-Bolyai University.

[12] Kaya Mehmet and James W. Fawcett. 2017. Identification of Extract Method Refactoring Opportunities through Analysis of Variable Declarations and Uses. *International Journal of Software Engineering and Knowledge Engineering* 27 (2017), 49–69. Issue 1.

[13] Iman Hemati Moghadam and Mel Ó Cinnéide. 2011. Code-Imp: A Tool for Automated Search-based Refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools (WRT '11).* ACM, New York, NY, USA, 41–44.

[14] Weifeng Pan, Bo Jiang, and Youyang Xu. 2013. Refactoring Packages of Object-oriented Software Using Genetic Algorithm Based Community Detection Technique. *International Journal of Computer Applications in Technology* 48, 3 (Oct. 2013), 185–194.

[15] A. Ananda Rao and K. Narendar Reddy. 2012. Identifying Clusters of Concepts in a Low Cohesive Class for Extract Class Refactoring Using Metrics Supplemented Agglomerative Clustering Technique. *CoRR* abs/1201.1611 (2012). http://arxiv.org/abs/1201.1611

[16] Singer J. Simons, C. and D.R. White. 2015. Search-based Refactoring: Metrics are Not Enough. In *Barros M., Labiche Y. (eds) Search-Based Software Engineering. SSBSE 2015. Lecture Notes in Computer Science*, Vol. 9275. Springer, Cham, 47–61.

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