

## Automated Refactoring of Object-Oriented Code Using Clustering Ensembles

**Timofey Bryksin**

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

**Alexey Shpilman**

Saint Petersburg National Research Academic  
University of the Russian Academy of Sciences  
JetBrains Research, Saint Petersburg, Russia  
alexey@shpilman.com

**Daniel Kudenko**

Department of Computer Science, University of York, United Kingdom  
Saint Petersburg National Research Academic University of the Russian Academy of Sciences  
JetBrains Research, Saint Petersburg, Russia  
daniel.kudenko@york.ac.uk

### Abstract

In this paper we are approaching the problem of automatic refactoring detection 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. An experimental validation of the proposed approach on an open source project is proposed. The obtained results illustrate that the introduced approach could be successfully used to improve existing integrated development environments, providing developers with one more tool to reduce complexity of their projects.

### Introduction

Refactoring is a well known tool helping to improve code readability and reduce its internal complexity, which make 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 complex 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 the code. Some of these metrics are very arguable since they are built based on opinions on what a good architecture is ((Simons and White 2015) even shows that refactoring metrics are weakly correlated with human judgment), but nevertheless a programmer can normally find some 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 refactorings for object-oriented projects. This allows to mitigate the problem of using just one single quality metric or approach implemented in each of the different clustering methods 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 (which is unique to our system) 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 is doable using IDEA’s internal refactoring tools).

### Related work

Detection of refactoring opportunities in software projects is a well-researched domain. Several techniques to detect design defects and appropriate refactorings have been presented: bayesian belief networks (Khomh et al. 2009), game theory (Bavota et al. 2010), multi-objective genetic programming (Mansoor et al. 2017), clustering ((Rao and Reddy 2012), (Marian 2014)) and a number of different search methods (for example, (Moghadam and Ó Cinnéide 2011), (O’Keeffe and Ó Cinnéide 2008) or (Harman and Tratt 2007)). Several papers present similar ideas to package-level refactorings ((Pan, Jiang, and Xu 2013), (Alkhalid, Alshayeb, and Mahmoud 2011), (Mahdavi, Harman, and Hiron 2003)).

Several tools exist that detect “code smells” and architectural defects like PMD<sup>1</sup>, iPlasma<sup>2</sup> or JDeodorant<sup>3</sup>. PMD is

<sup>1</sup>PMD static code analyzer: <https://pmd.github.io/>

<sup>2</sup>iPlasma code analyzer: <http://loose.upt.ro/reengineering/research/iplasma>

<sup>3</sup>JDeodorant Eclipse plugin: <https://marketplace.eclipse.org/content/jdeodorant>

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 in development process. JDeodorant is a plugin for Eclipse IDE, and it is able to suggest *Extract Method* and *Extract Class* refactorings.

Like JDeodorant the proposed 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 more diverse set of results.

## The approach

### Selected algorithms

During the literature review we have selected three algorithms that reported the highest percentage of justifiable refactorings: ARI and HAC from (Marian 2014) and CCDA from (Pan, Jiang, and Xu 2013).

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 metrics: Relevant Properties (RP), Depth in Inheritance Tree (DIT), Number of Children (NOC), Fan-In (FI) and Fan-Out (FO). RP for a method 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 consists of the class itself, all fields and methods from this class, all interfaces implemented and all classes extended by this class. So if an entity  $s_i$  is represented by a vector  $(s_{i1}, s_{i2}, s_{i3}, s_{i4}, s_{i5})$ , a semi-metric function shown in Formula 1 will measure dissimilarity between two entities from the system.

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

The algorithm initially places each class into a separate cluster and then tries to put each method to 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 together. 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. Then while changes can be done it calculates distances between all pairs of clusters, takes the minimum distance and merges these clusters together if it is less than 1. 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 package level. 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 edges between nodes are created when there is a dependency between the corresponding classes. Edges are weighted to specify strength of the dependency. The algorithm tries to detect communities within such a network: sets of nodes that have 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 the 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 this research CCDA was adopted to work at class level, redistributing methods between classes instead of redistributing classes between packages.

In addition to the three aforementioned algorithms this paper introduces two more. One of them is MRI (Modified ARI) which works like ARI, but it recalculates the Relevant Properties metric for each entity after moving a method to the closest cluster. That way MRI can calculate each new method movement taking into account all previous steps, while ARI uses only information on initial code structure for all steps.

Another algorithm introduced in this paper is called AK-Means (Advanced k-means). It combines ideas of hierarchical clustering and k-means: initially  $k$  methods are randomly selected and marked as centers of separate clusters. Each remaining method is then moved into the closest cluster (the *complete link* linkage method is used to calculate the distance). The initial number of classes is selected as the value of  $k$ , so this algorithm does not detect *Extract Class* and *Inline Class* refactorings. Due to the non-deterministic selection of initial methods this algorithm is being run multiple times (25 iterations were empirically selected) and the most popular partition is chosen as the result.

### Implementation details

All described algorithms were modified to handle movements of class fields along with methods, which made detection of *Move Field* refactorings also possible. To archive this in ARI, HAC and MRI the Relevant Properties metric for a field consists of the field itself, the class where the field is defined and all methods from the analysis scope, which access this field. Fields in CCDA and AKMeans are handled exactly as methods.

Several novel heuristics were introduced to ARI, HAC and MRI 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. Such dependencies are not

considered strong enough to move an entity 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 fields and methods (both static and not) and to the entity itself when it is added to its own Relevant Properties.

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 the minimum from 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 sum of weights of nonoverlapping entities plus the cardinality of the intersection of these sets.

This modification allows to assess strength of dependencies between entities and filter out refactorings that were clearly 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. It is calculated as follows:

- for ARI, MRI and HAC an auxiliary *difference* metric is calculated as difference between distances to first and second closest clusters. If an entity is decided to be moved into a cluster, the accuracy value of this refactoring is set to the minimum of  $5 * difference$  and 1.
- for CDDA and AKMeans 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 a cluster, this cluster's density value becomes the accuracy value of this refactoring.

The third heuristic concerns representation of results. Each algorithm produces a set of (*entity, target, accuracy*) refactorings. All sets are merged together and grouped by the *entity* value and each *accuracy* value is squared. If a particular refactoring (an *entity* and *target* pair) was suggested by more than one algorithm, their *accuracy* values are summed up into a *totalAccuracy* value. Then for each entity the refactoring with higher *totalAccuracy* value is selected as the most relevant one. Its final *accuracy* value will be  $\sqrt{totalAccuracy/n}$ , where  $n$  is a number of algorithms that suggested this refactoring. If this value is more than an empirically selected threshold of 0.6 then the refactoring is considered worthy.

## ArchitectureReloaded IDEA plugin

All five algorithms were implemented as a plugin for IntelliJ IDEA, the source code is available on GitHub<sup>4</sup>. The plugin allows to run the algorithms on the project code currently opened in the IDE, shows suggested refactorings in a tool window, allows to navigate to corresponding parts of the project and tries to apply selected refactorings automatically using IDEA's existing refactoring tools.

## Experimental evaluation

The algorithms were tested on a number of projects to see the relevance of suggested refactorings. One of them was JHotDraw 5.1 — an open-source project that is a well-known example for the use of design patterns and for good design and that was used for evaluation by the authors of ARI algorithm. In (Marian 2014) ARI is reported to find 20 refactorings, 11 of which the authors considered justifiable. Unfortunately no open implementation of this algorithm was provided, so we had to create our own. It suggested 32 refactorings, only 10 of which were mentioned in the original paper. The implementation with aforementioned heuristics suggested 10 refactorings, 8 of which were mentioned in the original paper and considered justifiable. The remaining 2 are considered incorrect since they are trying to move methods containing access to private fields.

The whole ensemble for JHotDraw 5.1 suggests 7 refactorings with an accuracy value more than 0.6, 5 of which in our opinion are justifiable:

- moving `DiamondFigure.polygon()` to `RectangleFigure` is justifiable since it is basically moving a method to a parent class, where it is also applicable.
- moving `FigureAttributes.read()` to `StorableInput` and `FigureAttributes.write()` to `StorableOutput` can not be done automatically since these methods access `FigureAttributes'` private field, but still are worthy to think about. For example, `read()` performs deserialization of the current object using a `StorableInput` stream object, so it makes sense to turn it into a factory method within `StorableInput` creating, initializing and returning a `FigureAttributes` object.
- moving `NumberTextFigure's` `getValue()` and `setValue()` to `TextFigure` is also moving methods to a parent class which can be performed automatically. All they do is call parent's methods, so within `TextFigure` these methods will work exactly the same. But these refactorings are justifiable only to some extent since these methods implement the feature specific to `NumberTextFigure` (handling the numbers) and it does not belong to the parent class. But despite such semantics being obvious to a human reader, there is no way for a clustering algorithm to get it.
- the remaining 2 refactorings (moving `FigureAttributes'` `get()` and `set()` methods to `AttributeFigure`) are clearly incorrect since all they do is access `FigureAttributes'` private field.

---

<sup>4</sup>ArchitectureReloaded plugin: <https://github.com/ml-in-programming/ArchitectureReloaded>

## 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 really judge whether this particular refactoring should be applied or not. Some of the refactorings that were suggested during the evaluation of the implemented algorithms could not have been applied as they were (for instance, moved methods accessed private fields or accessed other methods that were not possible to move), but still might give developer a hint of how to improve their code.

In a preliminary user evaluation of our refactoring plugin with five professional software developers we received positive feedback throughout concerning the results filtering. As mentioned before, only refactorings that got total accuracy value higher than 0.6 were initially shown (if there were no such refactorings, the top 10 were shown regardless of their accuracy values), but there was a slider that allowed to set any threshold value. It gave developers an opportunity to consider the most “popular” suggestions and review all others later if they were interested. And needless to say that if we want to create a tool that developers will use in everyday work, it should be integrated into their development environments.

## Conclusion and future work

Software refactoring can significantly improve quality of the code. For the last decade there has been a lot of research trying to suggest refactorings automatically, a lot of algorithms were proposed employing different techniques, but still very little of them found their way to industrial IDEs. In this paper we have proposed a way to combine existing knowledge in this field, creating ensembles of different algorithms. It has both academical and practical value: the implemented tool can be used to compare different refactoring identification algorithms and to test them on real life projects.

Although our approach produced promising results, there is room for further improvements. Future work will include:

- Further improvement of the algorithms: more experiments with weights for Relevant Properties metric and CCDA algorithm, selecting different  $k$  values for AKMeans algorithm, trying to move several entities together (for example, move a method and a private field that it accesses) etc.
- Identify and exploit the strengths of the individual refactoring identification algorithms to build a better ensemble.
- Optimize implemented tools for very large projects. Currently, a project consisting of a thousand classes is being processed for about 1-2 minutes, which could be considered too long for some users.
- Stronger IDE integration: for instance, efficiently perform background calculations and show the results using IDEA's code inspection annotations mechanism.

## References

- Alkhalid, A.; Alshayeb, M.; and Mahmoud, S. A. 2011. Software refactoring at the package level using clustering techniques. *IET Software* 5:274–286(12).
- Bavota, G.; Oliveto, R.; De Lucia, A.; Antoniol, G.; and Gueheneuc, Y.-G. 2010. Playing with refactoring: Identifying extract class opportunities through game theory. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM '10*, 1–5. Washington, DC, USA: IEEE Computer Society.
- Harman, M., and Tratt, L. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO '07*, 1106–1113. New York, NY, USA: ACM.
- Khomh, F.; Vaucher, S.; Guéhéneuc, Y.-G.; and Sahraoui, H. 2009. A bayesian approach for the detection of code and design smells. In *Proceedings of the 2009 Ninth International Conference on Quality Software, QSIC '09*, 305–314. Washington, DC, USA: IEEE Computer Society.
- Mahdavi, K.; Harman, M.; and Hierons, R. M. 2003. A multiple hill climbing approach to software module clustering. In *Proceedings of the International Conference on Software Maintenance, ICSM '03*, 315–. Washington, DC, USA: IEEE Computer Society.
- Mansoor, U.; Kessentini, M.; Maxim, B. R.; and Deb, K. 2017. Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal* 25(2):529–552.
- Marian, Z.-E. 2014. *Machine learning based software development*. Ph.D. Dissertation, Faculty of Mathematics and Computer Science, Babes-Bolyai University.
- Moghadam, I. H., and Ó Cinnéide, M. 2011. Code-imp: A tool for automated search-based refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools, WRT '11*, 41–44. New York, NY, USA: ACM.
- O’Keeffe, M., and í Cinnéide, M. 2008. Search-based refactoring for software maintenance. *J. Syst. Softw.* 81(4):502–516.
- Pan, W.; Jiang, B.; and Xu, Y. 2013. Refactoring packages of object-oriented software using genetic algorithm based community detection technique. *Int. J. Comput. Appl. Technol.* 48(3):185–194.
- Rao, A. A., and Reddy, K. N. 2012. Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique. *CoRR* abs/1201.1611.
- Simons, C., S. J., and White, D. 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*, volume 9275, 47–61. Springer, Cham.