# A review of Machine Learning Approaches for Move Method Refactoring Recommendation

Liviu- Stefan Neac su-Miclea
ICA 256/2

November 27, 2025

**Abstract**

Software maintenance is a costly part of a software's lifecycle, and refactoring is crucial for improving design quality and keeping code complexity between reasonable limits. The 'Move Method' refactoring is fundamental for addressing code smells like "Feature Envy" and improving class cohesion. However, manually identifying optimal refactoring targets in large-scale systems is a complex, time-consuming, and error-prone task. For this reason, Search-Based Software Engineering (SBSE) and Machine Learning (ML) techniques have been developed to automate the recommendation process. This report briefly reviews three methods meant to capture the evolution in the field. We discuss the transition from early approaches based on optimizing quality metrics to more recent methods based on more sophisticated ML, such as semantic analysis via topic modeling and modern deep learning models that learn from rich structural and semantic code representations. These strategies are compared, highlighting their theoretical foundations, benefits and limitations, and conclude on the future direction of intelligent refactoring systems.

## 1 Introduction

The phenomenon of software entropy is defined as the tendency of a software system to degrade as it keeps evolving. An effect of such degradation is the appearance of "code smells" – coding patterns that imply deeper design problems [7]. The process of improving internal code structure while preserving its external behavior is called refactoring, and is the principal approach in tackling the decay and manage technical debt.

One type of common and widely used refactoring actions is "Move Method", which involves moving a method from one class to a more suitable class, attempting to solve the "Feature Envy" code smell, where a method seems more interested in the members from another class than in its own. Ideally, 'Move Method' operation can significantly improve the system design as class cohesion increases by grouping related functionalities and coupling decreases by reducing dependencies between classes.

A posing challenge in refactoring is the need of automation. In large-scale projects, a method could potentially be moved to hundreds of other classes. Finding the correct class to move into is a complex search task [5]. Trying to move a method in a way that solely minimizes coupling might affect the conceptual coherence of the target class.

This report reviews the automatic solutions proposed in literature in order to solve this problem, from the classic search-based optimization to modern, data-driven machine learning techniques.

# 2 Motivation and Integration

The refactoring recommendation problem naturally fits in the field of Search-Based Software Engineering (SBSE). This field includes software engineering optimization problems that require exploring a large search space of possible solutions, looking for an optimal or near-optimal result according to a provided objective function. In case of refactoring, the search space consists of all possible refactor sequences of operations. The objective function is usually a formula that incorporates software quality metrics. The search problem for Move Method can be expressed as finding the pairs $(\text{method}, \text{targetClass})$ that result in the best improvement of overall quality of system design.

Early work on SBSE focused on finding sequences of refactoring actions to improve a design [5]. While being a powerful approach, it definitely has its limitations. The code metrics (LCOM, coupling) are heuristics that do not truly reflect the design quality. These methods are unaware of semantics or conceptual purpose of a method. A decision that is "good" in a way that properly optimize some metrics, might sometimes be perceived as illogical to the mind of human developers.

The lack of semantic context fueled the motivation to integrate Machine Learning approaches instead of relying on hand-crafted heuristics. ML models can learn high-quality non-trivial design patterns from numerous instances of existing source code. This creates a new mindset that is not searching for a score-optimal solution, but instead it predicts a solution that resembles one a human expert would apply [7].

# 3 Related Work

Recommending Move Method refactorings has been achieved in multiple ways, either by traditional metric systems, semantic considerations, and even hybrid representation learning.

## 3.1 QMove: Quality-Oriented and Metric-Based Search

The traditional SBSE approach is quality-oriented. These methods directly measure the impact of a potential Move Method operation on a set of well-defined quality metrics. An example is the work of Couto et al. [3], which proposes an approach (QMove) that recommends moves by evaluating a list of six empirically defined quality attributes (reusability, flexibility, understandability, functionability, extendibility, effectiveness) specific to a quality measuring system named QMOOOD [3]. The recommending system produces a sequence of refactoring moves. At each step, the algorithm finds the method move out of all the possible refactorings that produces the best improvement gain based on a fitness score. The evaluation reveals that QMove yields better results than similar approaches like JMove and JDeodorant, and is good at solving cases of Feature Envy, but is also more prone to producing false-positives, and is limited by the metrics used. All class memebers are treated equally regardless of their conceptual meaning relative to dependencies.

## 3.2 Methodbook: Semantic Analysis via Topic Modeling

Integrating semantic analysis came as signifincant breakthrough. The pioneering Methodbook system of Bavota et al. [2] was based on the insight that a method belongs to a class that shares the same conceptual topic. Methodbook uses Relation Topic Models (RTMs), with roots in statistical ML, to analyze the vocabulary used in comments and names of methods and variables. The entire codebase with classes and methods is modelled as a collection of documents, and infers their latent topics from a number of 75 topics the system works with [2]. A Move Method refactoring is then recommended if the conceptual topics of a method show greater similarity to the topics of another class compared to its source class. This approach succeeds in capturing the conceptual cohesion and can find more meaningful refactor opportunities than metric-based approaches. The number of suggestions by Methodbook was smaller than other approaches (JDeodorant), but it was argued that this is compensated by the quality of suggestions following the integration of textual information [2].

## 3.3 PathMove and RMove: Path-Based Code Representation

Newer ML techniques generally focus on learning rich representations of the involved entities. In case of code, instead of relying on topic models (such as bag of words), the models learn vector embeddings that capture deep structural and semantic properties.

The PathMove approach of Kurbatova et al. [6] (Fig. 1) uses path-based representations of code, just like similar techniques like code2vec [1]. Snippets of code are transformed into a collection of paths from its abstract syntax tree (AST) and convert it into a vector embedding representation. These paths capture fine-grained syntactic and structural relationships. Then, a machine learning model is trained on these representations to classify the (method, targetClass) pairs as good or bad moves, based on examples built from open-source projects [6]. Kurbatova et al. used a SVM classifier that has greater precision (37.6%) than JMove (18.7%) and JDeodorant (17.7%) on JMove dataset.

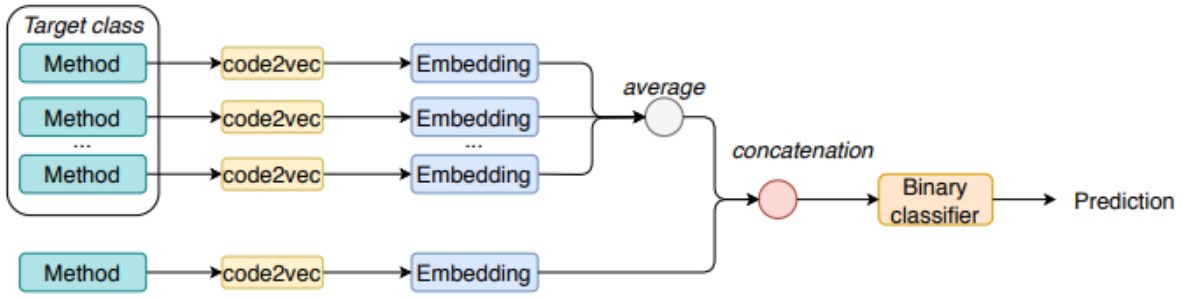


Figure 1: Path move classifier workflow

The RMove approach of Cui et al. [4] builds upon PathMove and proposes a state-of-the-art hybrid approach. It combines both structural and semantic representations (Fig. 2). It creates a method dependency graph, from which embeddings are extracted, and concatenated with features extracted from code in order to create the hybrid vector which is then passed to a ML classifier. A wide range of classifiers are tested in this work, including traditional ML (decision tree, SVM, logistic regression) and neural methods (CNN, LSTM, GRU). By feeding this combined, rich representation into a deep learning

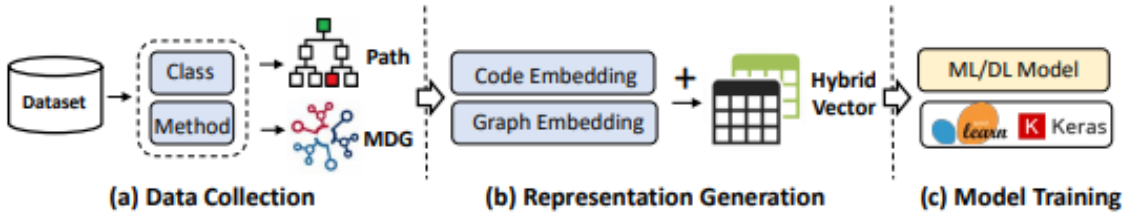


Figure 2: RMove method pipeline

model, RMove can achieve high performance (50% average precision on a real world datasets compared to PathMove 36%, JDeodorant 14.2% and JMove 15.2%), proving that the synergy of structure and semantics is crucial for this task.

# 4 Discussion

There is a clear trend in transitioning from metric-based search to representation learning approaches. While metric-based approaches [3] are explainable and directly optimize for known quality attributes, they suffer from a local optimum problem and a lack of conceptual understanding.

Topic modeling [2] was a major step forward, by employing semantics into the recommendation process. However, relying on natural language terms makes it vulnerable to poor variable namings or sparse comments, and the bag-of-words model disregards code structure.

Modern ML approaches [4, 6] represent the current state of the art. Their main advantage is the ability to learn complex, non-linear patterns from raw data. By observing millions of refactorings performed by humans, they can capture a nuanced understanding of good design than hand-crafted metrics. Particularly, the hybrid RMove model [4] confirms a key hypothesis: optimal refactoring decisions require both structural and semantic context.

However, these models have their limitations. First, they are data-hungry. They require massive, high-quality training datasets of labeled refactorings, which are expensive to mine and clean. Then, there is the black-box problem specific to the some classifier optimization: unlike a metric-based tool that can state that moving a certain method increases cohesion by a specific amount, a deep learning model's recommendation is opaque. This lack of explainability can be a major barrier to developer trust and adoption. Last, training deep models on code representations is computationally intensive compared to calculating simple metrics.

From a practical standpoint, the ground truth for evaluation is also a challenge. Modern papers [4, 6] evaluate their models by comparing their recommendations against refactorings actually performed by developers in subsequent commits. They are measured on precision (how many recommendations are correct) and recall (how many of the actual refactorings were found) and commonly compared to established approaches like JMove and JDeodorant. While this is the best available option, it assumes all human-performed refactorings are optimal and all missed ones are bad, which is not always the reality.

# 5 Conclusion and Future Work

The automated recommending Move Method refactoring has evolved from a straightforward search metric-based problem [3] to a sophisticated machine learning task. The field has progressed from optimizing structural metrics to modeling semantic cohesion [2], and finally to learning from rich, hybrid representations of code structure and semantics [4]. These advancements have significantly improved the accuracy and practical utility of refactoring tools.

However, significant challenges remain to be addressed. Future work will likely focus on explainability, by developing new models that can justify their recommendations in human-understandable terms, bridging the trust gap. Moreover, efforts might be directed to context-aware recommendation, integrating models into the IDE to provide recommendations that are not just correct in general, but relevant to the developer's current task.

Finally, largelanguage models (LLMs) trained on code seem to increase in popularity in the field of software engineering. These models may soon be able to move beyond recommending pre-defined operations and instead generate complex, multi-step refactorings [5] directly, based on high-level developer intent, further blurring the line between automated assistant and collaborative partner.

# References


[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 1–29. ACM, 2019.

[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, 2013.

[3] Christian Marlon Souza Couto, Henrique Rocha, and Ricardo Terra. A quality-oriented approach to recommend move method refactorings. In *Proceedings of the XVII Brazilian Symposium on Software Quality*, pages 11–20, 2018.

[4] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. Rmove: recommending move method refactoring opportunities using structural and semantic representations of code. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 281–292. IEEE, 2022.

[5] Erlend Kristiansen and Volker Stolz. Search-based composed refactorings. In *Norsk IKT-konferanse for forskning og utdanning*, 2014.

[6] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. Recommendation of move method refactoring using path-based representation of code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 315–322, 2020.

[7] Valentino Lenarduzzi, Davide Taibi, Michele Tufano, and Giuseppe Scanniello. A systematic literature review on machine learning in software refactoring. *Journal of Systems and Software*, 173:110887, 2021.