JMove: A Novel Heuristic and Tool to Detect Move Method Refactoring Opportunities

Ricardo Terra[1], Marco Tulio Valente[2], Sergio Miranda[2], Vitor Sales[2]

[1]Department of Computer Science, UFLA, Brazil
[2]Department of Computer Science, UFMG, Brazil
terra@dcc.ufla.br, {mtov, sergio.miranda, vitormsales}@dcc.ufmg.br

Abstract

This paper presents a recommendation approach that suggests Move Method refactorings using the static dependencies established by methods. This approach, implemented in a publicly available tool called JMove, compares the similarity of the dependencies established by a method with the dependencies established by the methods in possible target classes. We first evaluate JMove using 195 Move Method refactoring opportunities, synthesized in 10 open-source systems. In this evaluation, JMove precision ranges from 21% (small methods) to 32% (large methods) and its median recall ranges from 21% (small methods) to 60% (large methods). In the same scenario, JDeodorant, which is a state-of-the-art Move Method recommender, has a maximal precision of 15% (large methods) and a maximal median recall of 40% (small methods). Therefore, we claim that JMove is specially useful to provide recommendations for large methods. We reinforce this claim by means of two other studies. First, by investigating the overlapping of the recommendations provided by JMove and three other recommenders (JDeodorant, inCode, and Methodbook). Second, by validating JMove and JDeodorant recommendations with experts in two industrial-strength systems.

Keywords: Move method refactorings; Recommendation systems; Dependency sets; JMove; JDeodorant; Methodbook

1 Introduction

In object-oriented systems, classes encapsulate an internal state that is manipulated by methods. However, during software evolution, developers may inadvertently implement methods in incorrect classes [10]. On one hand, the causes of this design flaw are well-known, and include deadline pressures, complex requirements, or a partial understanding of the system's design. On the other hand, the consequences can be summarized in the form of a negative impact on the system's maintainability [13, 44]. The refactoring actions required to fix this design flaw are also well-documented. Basically, a Move Method must be applied to move the method from its current class

to the correct one [10, 42]. In fact, this refactoring is usually supported by automatic refactoring tools that are part of most modern IDEs. However, before applying this refactoring, maintainers must perform two program comprehension tasks: (a) detect the methods implemented in incorrect classes in the source code, and (b) determine the correct classes to receive these methods. Typically, such tasks are more complex because they require a global understanding of the system's design and implementation, which is a skill that only experienced developers have. However, despite their complexity, both tasks are usually performed without the support of any program analysis tools, at least in the case of mainstream IDEs.

To tackle this problem, in a previous conference paper we described a novel approach to recommend Move Method refactorings and a supporting tool called JMove [29]. This approach is centered on the following assumption: *methods in well-designed classes usually establish dependencies to similar types*. For example, suppose a hypothetical system where a `CustomerDAO` class is used to persist customers in a database. Typically, the dependency sets of the methods in this class include domain types (such as `Customer`) and also persistence related types (such as `Connection` and `SQLException`). Suppose also that one of such methods, called `getAllCustomers`, is inadvertently implemented in class `CustomerView`, responsible for user interface concerns. In this case, the dependency set of `getAllCustomers` does not contain dependencies to types like `Button`, `Label`, etc., which are usual in the methods from `CustomerView`. Therefore, the dependency set of `getAllCustomers` is more similar to the dependency sets in `CustomerDAO` than to the dependency sets in `CustomerView`. Based on this fact, our approach triggers a Move Method recommendation in this case, suggesting to move `getAllCustomers` from `CustomerView` to `CustomerDAO`.

In the presented paper, we extend our work in four directions. First, we evaluate JMove using a completely new sample of 195 manually synthesized Move Method refactoring opportunities, distributed over 10 well-known open-source systems. This sample includes both small and large methods, since JMove focuses on methods with a minimal number of dependencies (four dependencies, in our current setup). Second, we compare JMove not only with JDeodorant [42], but also with inCode [16, 15], which is a commercial quality assessment system. Third, we report an investigation on the overlapping of the recommendations provided by JMove, JDeodorant, inCode, and Methodbook [3, 24] in two open-source systems. Fourth, we report a study on the relevance of the recommendations provided by JMove—and also by JDeodorant and inCode—according to experts in two industrial-strength systems.

Essentially, we summarize our results as follows:

- *Precision:* JMove precision ranges from 21% (small methods) to 32% (large methods). JDeodorant produces around twice more recommendations than JMove, with a precision of 5% (small methods) and 15% (large methods). inCode produces no recommendations.

- *Recall:* JMove median recall doubles from small methods (21%) to large ones (60%). JDeodorant median recall is 40% for small and 35% for large methods. inCode recall is very low, just

10% for large methods. Furthermore, JMove's median recall reaches 79% on large methods, when identifying only the recommended method (but not the target class) is needed.

- *Overlapping with other tools:* 70% of the correct recommendations missed by JMove, but recommended by JDeodorant and Methodbook are in methods with less than four dependencies. This result reinforces the observation that JMove provides its best results for large methods.

- *Evaluation by experts:* When considering a classification performed by expert developers with real instances of misplaced methods, JMove does not provide recommendations for a system that does not have methods implemented in incorrect classes, while JDeodorant and inCode trigger 43 and 50 (false) recommendations, respectively. In a second system, JMove achieves a precision of 24%, against 6% and 5% from JDeodorant and inCode, respectively.

The remainder of this paper is organized as follows. Section 2 briefly presents previous research on strategies for recommending Move Method refactorings. Section 3 details our recommendation approach, including its main algorithms and functions. The section also presents a detailed example and a supporting tool. Section 4 presents the evaluation using 195 synthesized Move Method opportunities, from 10 open-source systems. Section 5 reports a comparative study of the recommendations provided by JMove, JDeodorant, inCode, and Methodbook in two open-source systems. Section 6 conducts a study with expert developers. Section 7 concludes the paper.

2 Related Work

Although it is possible to manually refactor a system, tool support is crucial in refactoring tasks [17]. Basically, there are two types of tools that can guide developers on refactoring activities: automatic and semi-automatic tools. Automatic refactoring and remodularization tools do not require the developer intervention [21, 20, 12, 1]. For example, Bunch [19] is a system that produces a new modularization for a system, which is generated by maximizing the value of an objective function that rewards highly cohesive modules that are not excessively coupled. By contrast, in semi-automatic tools, developers feedback is required to identify which parts of the software needs to be refactored or to select which refactorings should be applied. Recommendation systems can be considered as examples of semi-automatic tools. These systems are software applications that aim to support users in their decision-making while interacting with large amounts of information. Specifically, Recommendation Systems for Software Engineering (RSSEs) should provide potentially valuable information for software engineering tasks in a given context [28].

Since JMove aims to recommend Move Method refactorings, we discuss in this section three RSSEs with this same purpose: (a) JDeodorant, which is the system closer to our approach; (b) inCode, which is a commercial tool to identify design flaws; (c) Methodbook, which implements a solution based on Relational Topic Model to search for Move Method refactoring opportunities.

Finally, we also present other approaches to identify refactoring opportunities, mainly related to methods implemented in incorrect classes.

2.1 JDeodorant

Proposed by Tsantalis and Chatzigeorgiou, JDeodorant [42, 8] follows a classic heuristic to detect Move Method refactoring opportunities: a method m should be me moved to a class C' when m accesses more services from C' than from its own class C. However, JDeodorant includes two major improvements to this heuristic. First, the system defines a rich set of Move Method refactoring preconditions to check whether the refactoring recommendations can be applied, and preserve behavior and the design quality. For instance, there are preconditions to avoid *compilation errors* (e.g., a precondition that avoids moving methods including a `super` invocation), to assure *behavior-preservation* transformations (e.g., a precondition that avoids moving `synchronized` methods), and to preserve the *system's internal quality* (e.g., a precondition that avoids moving methods that are highly coupled to class attributes). Second, to avoid an explosion in the number of false positives, JDeodorant defines an *Entity Placement* metric to evaluate the quality of a possible Move Method recommendation. This metric is used to evaluate whether a recommendation reduces a system-wide measurement of coupling and at the same time improves a system-wide measurement of cohesion. On one hand, coupling is defined by the Jaccard distance between the class itself and outer entities (i.e., entities that do not belong to the class). The larger the Jaccard distance, the lower the coupling. On the other hand, cohesion is defined by the Jaccard distance between the class itself and inner entities. The smaller this distance, the higher the cohesion.

JDeodorant was evaluated in two main ways. First, the tool's authors applied the recommendations provided by JDeodorant on two open source projects (JEdit and JFreeChart) and measured the impact of these operations on coupling and cohesion. They concluded that the application of the recommended Move Method refactorings, in general, reduced coupling and increased cohesion, as expected. In a second evaluation, an independent designer assessed the conceptual integrity of the refactorings recommended by JDeodorant in a scientific software. He agreed in 8 out of 10 refactoring suggestions. Although JDeodorant was initially designed to identify Move Method refactoring opportunities, the tool has been recently extended to also identify Extract Method [43] and Extract Class refactorings [9].

2.2 inCode

inCode is a commercial tool that relies on a set of metric-based strategies to capture deviations from established design principles, including the detection of methods displaying a Feature Envy behavior [16]. The proposed strategies allow developers to directly localize classes or methods affected by a particular design flaw rather than having to infer the problem from a large set of metric values [41]. Fundamentally, a detection strategy is based on two mechanisms: filtering (to

reduce the initial data set) and composition (to correlate the interpretation of multiple metrics). As an example, the following strategy is used to infer that m is implemented in an incorrect class:

$$(\mathbf{ATFD}(m), \mathit{HigherThan}(\mathit{FEW})) \ \wedge \ (\mathbf{LAA}(m), \mathit{LessThan}(\mathit{ONE\ THIRD})) \wedge \\ (\mathbf{FDP}(m), \mathit{LessOrEqualThan}(\mathit{FEW}))$$

where ATFD (Access to Foreign Data) measures the number of distinct attributes the measured element access, LAA (Locality of Attribute Accesses) measures the relative number of attributes that a method accesses on its class, and FDP (Foreign Data Providers) measures the number of classes the accessed attributes belong to. The *FEW* and *ONE THIRD* thresholds are automatically set up by inCode without the user knowledge. Actually, determining "correct" threshold values is one of the major limitations of metric-based approaches and it is usually guided by empirical studies, past experiences, or benchmark data [14, 18, 23].

The detection strategies implemented by inCode to detect Feature Envy instances were evaluated in a large telecommunication system, with two versions (SV1 and SV2) [16]. SV2 is a re-engineered and enhanced version of SV1. They assume that a flaw detected in SV1 is a false positive if it reoccurs in SV2; otherwise it is a real flaw. In this way, they detected 40 suspected Feature Envy instances in SV1 and 25 were classified as real flaws, resulting in an accuracy of 63%.

2.3 Methodbook

Methodbook [3, 24] searches for Move Method refactoring opportunities using a technique called Relational Topic Model (RTM) [5]. This technique is used, for example, by Facebook to analyze users profiles and to suggest new friends or groups. Likewise, in the Methodbook approach, methods and classes play the role of users and groups, respectively. Moreover, methods' bodies are analogous to profiles in Facebook and contain information about structural (e.g., method calls) and conceptual relationships (e.g., comments) with other methods. Therefore, Methodbook would recommend moving a given method m to the class C' that has the largest number of m's "friends". This recommendation is based on the conjecture that the higher the number of friends placed in the same class, the higher its quality in terms of cohesion and coupling.

The authors initially evaluated Methodbook using six systems to check whether the tool can help to improve quality metrics. They also report a second study with eighty developers that were asked to evaluate the refactorings suggested by Methodbook for two systems. In Section 5, we compare the results produced by our approach with the recommendations produced by Methodbook in this second study. Finally, it is worth mentioning that RTM has also been used to recommend refactoring operations aiming at moving classes to more suitable packages [2].

2.4 Other Approaches

O'Keeffe and Ó Cinnéide proposed a search-based software maintenance tool that relies on search algorithms, such as Hill Climbing and Simulated Annealing, to suggest six inheritance-related refac-

torings [22]. However, they do not handle methods located in incorrect classes. Similarly, Seng et al. proposed a search-based approach that relies on an evolutionary algorithm and simulated refactorings to provide a list of refactorings that may lead to a system's structure presenting better metric values [31]. Based on the premise of the mature design of JHotDraw, the authors misplaced ten randomly selected methods and performed their approach, which could move back nine methods. We do not compare our work with Seng et al.'s approach because it does not include a public implementation. Palomba et al. propose HIST, an approach that uses association rule mining on version histories to detect code smells [26]. For each smell, they define a heuristics that relies on association rules for detecting bad smells. For Feature Envy, the proposed heuristic marks methods that change more often with methods in another class than with methods of their current class. Therefore, HIST adopts an evolutionary definition of feature envy. Since there is not a unique and established definition for this bad smell, throughout this paper we use the term Feature-Envy instances to denote methods implemented in incorrect classes; specifically, we do no use a strict definition of Feature-Envy that requires the focus of the Envy to be only data.

The recommendation approach described in this paper was inspired by our previous work on architecture conformance and repair [39, 38]. More specifically, we proposed a recommendation system, called ArchFix [40], which supports 32 repair recommendations to remove violations raised by architecture conformance checking approaches. Five of such refactorings include a recommendation to move a method to another class, which is inferred using the set of dependencies established by the source method and the target class. However, in ArchFix the methods implemented in incorrect classes are given as an input to the recommendation algorithm (and represent an architectural violation). On the other hand, in the current work, our goal is exactly to discover such methods automatically. Moreover, we also target methods that do not necessarily denote an architectural violation. Inspired by our previous work, Silva et al. proposed an approach that relies on dependency sets to identify and rank Extract Method refactorings [32]. Basically, this approach aims to recommend new methods that include structural dependencies that are rarely used by the remaining statements in the original method. The evaluation using a sample of 81 Extract Method opportunities generated for JUnit and JHotDraw achieved a precision of 48% and 38%, respectively.

3 Proposed Approach

Our approach detects methods located in incorrect classes and then suggests moving such methods to more suitable ones. More specifically, we first evaluate the set of static dependencies established by a given method m located in a class C, as illustrated in Figure 1. After that, we compute two similarity coefficients: (a) the average similarity between the set of dependencies established by method m and by the remaining methods in C; and (b) the average similarity between the dependencies established by method m and by the methods in another class C_i. If the similarity measured in the step (b) is greater than the similarity measured in (a), we infer that m is more

similar to the methods in C_i than to the methods in its current class C. Therefore, C_i is a potential candidate class to receive m.

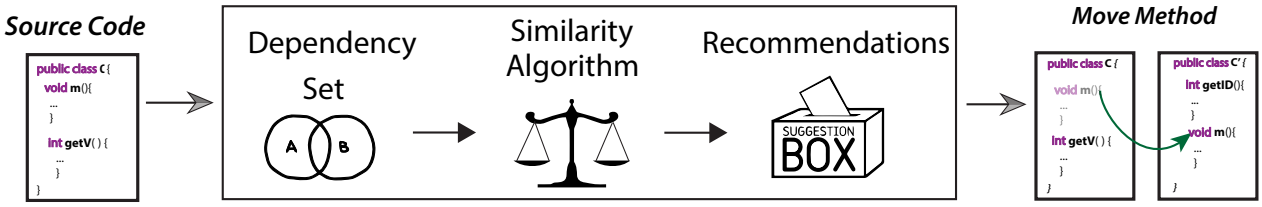


Figure 1: Proposed approach

In the remainder of this section, we describe the proposed recommendation algorithm (Section 3.1), the similarity functions that play a central role in this algorithm (Section 3.2), and the strategy we designed to decide the most suitable class to receive a particular method (Section 3.3). We also empirically assess our defined thresholds through a calibration experiment (Section 3.4). Moreover, we describe a supporting tool (Section 3.5) and present four examples of Move Method refactorings suggested by JMove and two state-of-the-art tools (Section 3.6). Finally, we document the limitations of our approach (Section 3.7).

3.1 Recommendation Algorithm

Algorithm 1 presents the proposed recommendation algorithm. Assume a system S with a method m implemented in a class C. For all class $C_i \in S$, the algorithm verifies whether m is more similar to the methods in C_i than to the methods in its original class C (line 6). In the positive cases, we insert C_i in a set T that contains the candidate classes to receive m (line 7). Finally, we search in T for the most suitable class to receive m (line 10). In case we find such a class C', we make a recommendation to move m to C' (line 11).

Algorithm 1 Recommendation algorithm

Input: Target system S
Output: Set with Move Method recommendations

```
1:  Recommendations ← ∅
2:  for all method m ∈ S do
3:      C = get_class(m)
4:      T ← ∅
5:      for all class C_i ∈ S do
6:          if similarity(m, C_i) > similarity(m, C) then
7:              T ← T ∪ {C_i}
8:          end if
9:      end for
10:     C' = best_class(m, T)
11:     Recommendations ← Recommendations ∪ { move(m, C') }
12: end for
```

This algorithm relies on two key functions: (a) $similarity(m, C)$ that computes the average similarity between method m and the methods in a class C; and (b) $best\_class(m, T)$ that receives a set T of candidate classes to receive m and returns the most suitable one. These functions are detailed in the following sections.

3.2 Similarity Function

Our approach relies on the set of static dependencies established by a method m to compute its similarity with the methods in a class C, as described in Algorithm 2. Initially, we compute the similarity between method m and each method m' in C (line 4). In the end, the similarity between m and C is the arithmetic mean of the similarity coefficients computed in the previous step. In this algorithm, $NOM(C)$ denotes the number of methods in a class C (lines 8 and 10).

Algorithm 2 Similarity Algorithm

Input: Method m and a class C
Output: Similarity coefficient between m and C

```
1: sim ← 0
2: for all method m' ∈ C do
3:     if m ≠ m' then
4:         sim = sim + meth_sim(m, m')
5:     end if
6: end for
7: if m ∈ C then
8:     return sim / [NOM(C) − 1]
9: else
10:    return sim / NOM(C)
11: end if
```

The key function in Algorithm 2 is $\texttt{meth\_sim}(\texttt{m}, \texttt{m}')$, which computes the similarity between the sets of dependencies established by two methods (line 4). We decided for the use of the *Jaccard* coefficient, since it has been used in similar approaches [33, 42, 8, 43], defined as:

$$\texttt{meth\_sim}(\texttt{m}, \texttt{m}') = \frac{a}{a+b+c} \tag{1}$$

where

- $a = \mid Dep(m) \bigcap Dep(m') \mid$
- $b = \mid Dep(m) - Dep(m') \mid$
- $c = \mid Dep(m') - Dep(m) \mid$

In this definition, $Dep(m)$ is a set with the dependencies established by method m. This set includes the types the implementation of m establishes dependency with. More specifically, we consider the following dependencies:

- *Method calls*: if method m calls another method m', the class of m' is added to $Dep(m)$.
- *Field accesses*: if method m reads from or writes to a field f, the type of f is added to $Dep(m)$. When f is declared in the same class of m, then the class itself is also added to $Dep(m)$.
- *Object instantiations*: if method m creates an object of a type C, then C is included in $Dep(m)$.

- *Local declarations*: if method m declares a variable or formal parameter v, the type of v is included in $Dep(m)$.
- *Return types*: the return type of m is added to $Dep(m)$.
- *Exceptions*: if method m can raise an exception E or if method m handles E internally, then the type of E is added to $Dep(m)$.
- *Annotations*: if method m receives an annotation A, then the type of A is included in $Dep(m)$.

When building the dependency sets we ignore the following types: (a) primitive types; and (b) types and annotations from `java.lang` and `java.util` (such as `String`, `HashTable`, `Object`, and `SupressWarnings`). Since virtually all classes may establish dependencies with these types, they do not actually contribute for the measure of similarity. This decision is quite similar to the one followed by text retrieval systems that exclude stop words because they are not helpful in describing the content of a document.

For the sake of clarity, we omitted from Algorithm 2 a test that discards three kinds of methods when calculating the similarity of dependency sets:

- A methods m that is the only method in its class because our approach is based on the similarity between a given method and the remaining methods in the class.
- A method m whose dependency set has less than four dependencies because few dependencies are more subjected to imprecise or spurious similarity measures. We defined this threshold after experimental tests where we found that lower thresholds would contribute to an explosion in the number of recommendations (refer to Section 3.4).
- A method m without parameters and with a single statement, which is a `return` statement. Based on a heuristic proposed by Seng et al. [31], the goal is to filter out *getter* methods, which by their very nature are rarely implemented in incorrect locations. The same happens with *setter* methods, but they are filtered out by the Move Method preconditions, as described in the next section.

3.3 Target Class Selection

Assume that T is now a list with classes $C_1, C_2, \ldots, C_n$ that are more similar to a method m than its current class C, as computed by Algorithm 1. Assume also that the classes in T are in descending order by their similarity with m—i.e., most similar classes first—as computed by Algorithm 2.

To reduce the chances of false positives, a move recommendation is *not* created when: (i) T has less than three classes, and (ii) the difference between the similarity coefficient value of C_1 (the first class in the list) and C (the original class of m) is less than or equal to 25%. In such cases, we

consider that the difference between the dependencies established by C_1 and C is not discrepant enough to recommend a move operation.

On the other hand, when such conditions do *not* hold, a recommendation $move(m, C_i)$ is created for the first class $C_i \in T$ that satisfies the preconditions of the Move Method refactoring ($1 \leq i \leq n$). Basically, as usual in the case of refactorings [25, 10], a Move Method has its application conditioned by a set of preconditions, fundamentally to ensure that the program's behavior is preserved after the refactoring. For example, the target class C_i should not contain a method with the same signature of method m. When such preconditions are not satisfied by a pair (C, C_i), we automatically verify the next pair (C, C_{i+1}). No recommendation is returned when the refactoring preconditions fail for all pair of classes in T.

We currently rely on preconditions of the Move Method automatic refactoring engine provided by the Eclipse IDE. However, it is well-known that Eclipse implements weak preconditions for some refactorings [35, 30, 34]. For this reason, we strengthened the Eclipse preconditions by supporting the following five preconditions originally proposed by Tsantalis et al. [42]:

- The target class should not inherit a method having the same signature of the moved method.
- The method to be moved should not override an inherited method in its original class.
- The method to be moved should not be synchronized.
- The method to be moved should not contain assignments to its original class fields (including inherited fields).
- The method to be moved should have a one-to-one relationship with the target class.

3.4 Threshold Calibration

As stated in the previous sections, JMove relies on two thresholds to reduce the number of false positives: (i) a move recommendation is *not* created for methods whose number of dependencies is less than four; (ii) a move recommendation is *not* created when the number of candidate classes is less than three and the difference between the similarity coefficient value of C_1 (the top recommended class) and C (the original class where the method is) is less than 25%.

We defined these thresholds after a preliminary empirical assessment using JHotDraw, an open-source system with a stable and well-documented architecture. Our primary assumption for using JHotDraw in this calibration is that *most JHotDraw's methods are likely to be in their proper classes*, since JHotDraw is developed and maintained by a small number of expert developers. Therefore, the rationale behind our calibration experiment is to set the thresholds up with the less restrictive values that recommend fewer recommendations.

Number of dependencies: We ran JMove in the JHotDraw system, switching the minimum number of dependencies a method must have to be analyzed. Table 1 reports the results. We

chose less than four dependencies because this configuration provides a small number of recommendations (15) but at the same time the number of methods analyzed is still representative (2,171).

Table 1: Number of dependencies threshold

dependencies	recommendations	analyzed methods
< 1	36	6,749
< 2	31	5,164
< 3	22	3,400
$\mathbf{< 4}$	**15**	**2,171**
< 5	7	1,422
< 6	6	985
< 7	4	694
< 8	2	513
< 9	1	340
< 10	1	235

Other thresholds: We re-ran JMove in the JHotDraw system, switching the minimum number of candidate classes as well as the minimum difference between C_1 (the top recommended class) and C (the original class where the method is). In this test, we only consider methods with at least four dependencies (as concluded in the previous calibration). Figure 2 reports the results. First, it is important to highlight the importance of threshold W. The number of recommendations drops from 15 to 9 recommendations when we enable this threshold, setting its value as 25% or 50%. Moreover, threshold C has also a positive impact in the results. When it is disabled ($C < 0$), we always have the highest number of recommendations (15 recommendations). Therefore, we decided to set these thresholds as follows: less than three candidates (threshold C) and the difference less than 25% (threshold W) since it is the less restrictive pair value (C, W) that achieves the minimal number of recommendations in JHotDraw (nine recommendations).

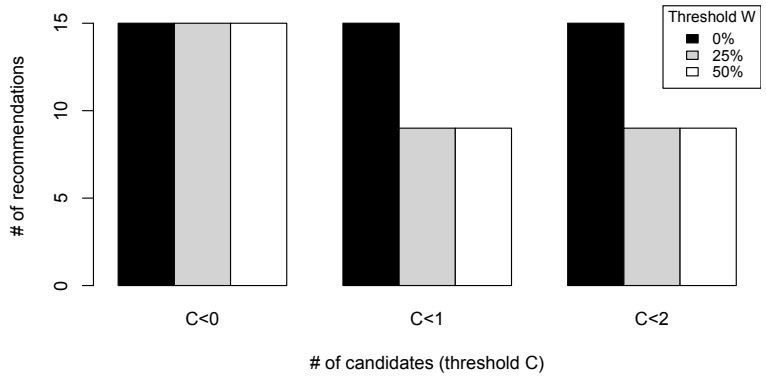


Figure 2: Calibrating thresholds about the number of candidate classes (threshold C), and the difference between top-candidate class and the original class of a method (threshold W)

3.5 Tool Support

We implemented a prototype tool, called JMove, that supports our approach. Basically, JMove is an Eclipse plug-in that implements Algorithms 1 and 2. Figure 3 illustrates the tool's interface using our previous running example, where method `getAllCustomers` from the persistence layer was purposely implemented in class `CustomerView`. As can be noticed, JMove adds a report panel to the IDE, which is used to display the Move Method recommendations.

Figure 4 illustrates the main modules from JMove's architecture. First, module *Dependency Set Extraction* is used to extract the dependency sets that characterize the methods in our approach. Second, module *Similarity Measurement* implements Algorithms 1 and 2. Third, module *Recommender System* implements function $best\_class(m, T)$.

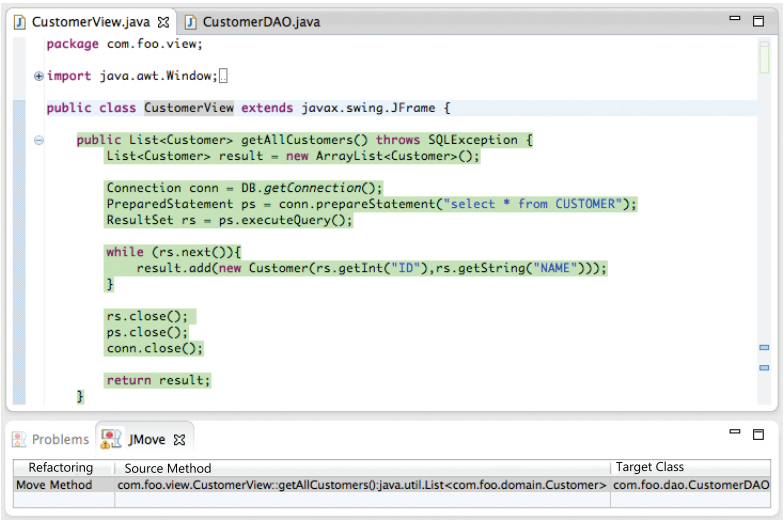


Figure 3: JMove's interface

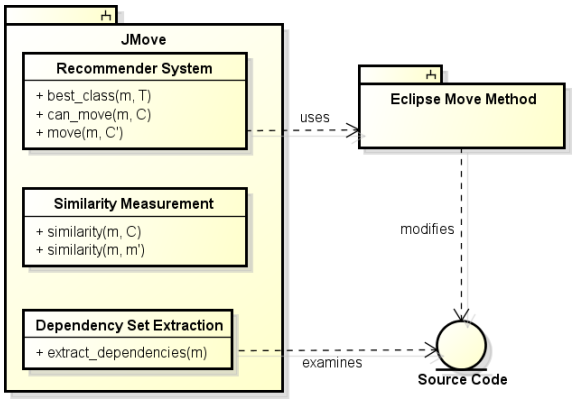


Figure 4: JMove's architecture

3.6 Examples

In this section, we present four examples of Move Method refactorings suggested in JHotDraw by JMove and two state-of-the-art tools—JDeodorant and inCode—we previously introduced in Sections 2.1 and 2.2.

Example #1: Code 1 shows our first example in which method `calculateLayout2`—a large method moved from class `LocatorLayouter`—does not access any service from its class `AttributeKey`. As a result, the dependency set of `calculateLayout2` is very different from the dependency sets of the other methods in `AttributeKey`, as follows:

$$Similarity(calculateLayout2, AttributeKey) = 0.03$$

```
1:public class AttributeKey<T> implements Serializable {
    ... piece of code
2:  Double calculateLayout2(LocatorLayouter locLayouter,
3:                      CompositeFigure compositeFigure, Double anchor, Double lead) {
4:      Double bounds = null;
5:      for (Figure child: compositeFigure.getChildren()){
6:          Locator loc = locLayouter.getLocator(child);
7:          Double r;
8:          if (loc == null) {
9:              r = child.getBounds();
10         } else {
11:             Double p = loc.locate(compositeFigure);
12:             Dimension2DDouble d = child.getPreferredSize();
13:             r = new Double(p.x, p.y, d.width, d.height);
14:         }
15:         if (!r.isEmpty()) {
16:            ... piece of code
17:         }
18:     }
19:     return (bounds == null) ? new Double() : bounds;
20:  }
21:}
```

Code 1: First Move Method Example (JHotDraw)

In fact, this method is more similar to the methods in class `LocatorLayouter`. To illustrate, we report the similarity between `calculateLayout2` and the three methods in `LocatorLayouter`:

$$\begin{aligned} meth\_sim(calculateLayout2, layout) &= 1.00 \\ meth\_sim(calculateLayout2, calculateLayout) &= 0.5 \\ meth\_sim(calculateLayout2, getLocator) &= 0.43 \end{aligned}$$

As result, we have that:

$$Similarity(calculateLayout2, LocatorLayouter) = 0.64$$

Due to this high similarity, JMove correctly recommends moving `calculateLayout2` back to its original class `LocatorLayouter`. JDeodorant does not make a recommendation to move this method back. Basically, in the case of *getter* methods, as the `getLocator` call in line 6,

JDeodorant considers that the method envies not the target object type (`LocatorLayouter`) but the type returned by the call (`Locator`). However, it is not possible to move `calculateLayout2` to `Locator` because the refactoring preconditions fail in this particular case. Finally, inCode also fails in suggesting to move `calculateLayout2` because the ATFD (Access to Foreign Data) metric considers only accesses to external fields, e.g., direct access to fields from external classes or via *getter* methods. In `calculateLayout2`, nonetheless, only methods from the method's own class (`LocatorLayouter`) are called.

Example #2: Code 2 shows the second example discussed here. Similar to the previous example, method `fireAreaInvalidated2` does not access any service from class `DrawingEditorProxy`. However, it calls three methods from `AbstractTool` (lines 4-6) and the Move Method preconditions are satisfied. For this reason, JDeodorant infers that `AbstractTool` is the most suitable class for the method.

```
1:public class DrawingEditorProxy extends AbstractBean
2:   implements DrawingEditor {
     ... piece of code
3:   void fireAreaInvalidated2(AbstractTool abt, Double r){
4:       Point p1 = abt.getView().drawingToView(...);
5:       Point p2 = abt.getView().drawingToView(...);
6:       abt.fireAreaInvalidated(
7:       new Rectangle(p1.x, p1.y, p2.x - p1.x, p2.y - p1.y));
8:   }
9:}
```

Code 2: Second Move Method Example (JHotDraw)

JMove also suggests to move `fireAreaInvalidated2` back to `AbstractTool` because the method is more similar to this class than to its current class, as indicated by the similarity function results:

$$Similarity(fireAreaInvalidated2,\ DrawingEditorProxy) = 0.03$$
$$Similarity(fireAreaInvalidated2,\ AbstractTool) = 0.17$$

Example #3: Code 3 shows an example in which all tools successfully recommend moving a method to its original class. As in the previous examples, method `setEditor`—moved from `SVGDrawingPanel`—does not access any method or field from its current class `ViewToolBar`. However, the method accesses 13 different attributes from class `SVGDrawingPanel`.

For JDeodorant, a method m envies a class C' when m accesses more attributes and methods from C' than from its own class. Therefore, JDeodorant correctly infers that class `SVGDrawingPanel` is the most suitable class for the method. inCode also successfully identified `setEditor` as a misplaced method. Particularly, the three rules that compose the detection strategy described in Section 2.2 are satisfied in this case: (i) the method accesses 13 different attributes from the external class `SVGDrawingPanel`; (ii) the method does not access any service from its current

```
1:public class ViewToolBar extends AbstractToolBar {
    ... piece of code
2:  public void setEditor(SVGDrawingPanel svg, DrawingEditor newValue) {
3:    ... piece of code
7:      svg.editor = newValue;
8:      if (newValue != null) {
9:          newValue.add(svg.view);
10:     }
11:     svg.creationToolBar.setEditor(svg.editor);
12:     svg.fillToolBar.setEditor(svg.editor);
13:     svg.strokeToolBar.setEditor(svg.editor);
14:     svg.actionToolBar.setUndoManager(svg.undoManager);
15:     svg.actionToolBar.setEditor(svg.editor);
16:     svg.alignToolBar.setEditor(svg.editor);
17:     svg.arrangeToolBar.setEditor(svg.editor);
18:     svg.fontToolBar.setEditor(svg.editor);
19:     svg.figureToolBar.setEditor(svg.editor);
20:     svg.linkToolBar.setEditor(svg.editor);
21:     ... piece of code
31: }
32:}
```

Code 3: Third Move Method Example (JHotDraw)

class; and (iii) only attributes of class `SVGDrawingPanel` are accessed by the method. JMove also suggests to move `setEditor` back to `SVGDrawingPanel` because the method is more similar to `SVGDrawingPanel` than to any other class in the system.

Example #4: In Code 4, method `loadDrawing` does not access any service from its new class `ProgressIndicator`. However, it calls four methods from its original class `SVGApplet` (lines 3–5) and the Move Method preconditions are satisfied. For this reason, JDeodorant correctly infers that `SVGApplet` is the most suitable class for the method. inCode fails in recommending to move `loadDrawing` because it only calls methods (not fields) from `SVGApplet`. Finally, JMove also fails because the method is not similar in terms of dependencies to the other methods in `SVGApplet`. Due to its specific task of loading a `Drawing`, the dependency set of the method contains many particular types, such as `URL` (line 5), `URLConnection` (line 6), `BufferedInputStream` (line 10), and `IOException` (line 12). For this reason, it has a low similarity to its original class:

$$Similarity(loadDrawing,\ SVGApplet)\ = 0.07$$

3.7 Limitations

JMove does not provide recommendations for methods that have less than four dependencies and for methods that are the single methods in their classes (as explained in Section 3.2). Furthermore, we do not recommend to move methods that do not attend the refactoring preconditions. This decision is based on the fact that the moving operation, in this case, typically requires a more complex restructuring both in the source and in the target classes. Finally, we do not provide suggestions to move fields since it is rare to observe fields declared in incorrect classes [42].

```
1:public class ProgressIndicator extends javax.swing.JPanel {
    ... piece of code
2:public Drawing loadDrawing(SVGApplet svgApplet) throws IOException {
3:  Drawing drawing = svgApplet.createDrawing();
4:  if (svgApplet.getParameter("datafile") != null) {
5:     URL url = new URL(svgApplet.getDocumentBase(), svgApplet.getParameter("datafile"));
6:     URLConnection uc = url.openConnection();
    ... piece of code
10: BufferedInputStream bin = new BufferedInputStream(in);
11: bin.mark(512);
    ... piece of code
12: IOException formatException = null;
    ... piece of code
13: return drawing;
14: }
15:}
```

Code 4: Fourth Move Method Example (JHotDraw)

4 Evaluation with Open-Source Systems

In this section we evaluate our recommendation approach in 10 open-source systems, using a synthesized sample of methods implemented in incorrect classes. We compare JMove results with the ones produced by JDeodorant and inCode.

4.1 Research Questions

This first study aims to answer the following research questions:

- ***RQ #1*** – How does JMove compare with related tools in terms of precision?
- ***RQ #2*** – How does JMove compare with related tools in terms of recall?
- ***RQ #3*** – How does JMove compare with related tools in terms of runtime performance?
- ***RQ #4*** – How does JMove perform when configured with different set similarity coefficients?

We compare our results with JDeodorant and inCode. We selected these tools for the following reasons: (a) JDeodorant is a well-known system for identifying refactoring opportunities; and (b) inCode is a commercial solution for identifying design flaws.

4.2 Study Design

In this section, we present the dataset used in our evaluation and the methodology we followed to generate our gold sets (i.e., the methods implemented in incorrect classes).

4.2.1 Dataset

We evaluate JMove using the Qualitas.*class* Corpus [37]. This corpus is a compiled version of 111 systems originally included in the Qualitas Corpus [36], which solely provides the source code of the systems. However, for evaluating tools that depend on Abstract Syntax Tree (AST), as the one proposed in this paper, we need to import and compile the source code, which is not trivial in the case of systems with complex external dependencies.

Our dataset includes nine systems from the Qualitas.*class* Corpus, which were randomly selected among the systems in this corpus. We also deliberately include JUnit in the dataset, which is a system with a well-defined and mature design, designed by expert developers [27]. Table 2 presents the ten systems considered in the study, including their names, version, number of classes (NOC), number of methods (NOM), and size in lines of code (LOC).

Table 2: Target Systems

System	**Version**	**NOC**	**NOM**	**LOC**
Ant	1.8.2	1,290	12,641	127,507
Derby	10.9.1.0	2,995	34,515	651,118
DrJava	r5387	1,866	8,399	89,477
JFreeChart	1.0.13	649	10,990	143,062
JGroups	2.10.0	1,211	7,324	96,325
JTopen	7.8	2,054	22,276	342,032
JUnit	4.10	171	801	6,580
MvnForum	1.2.2	273	7,743	105,328
Lucene	4.2.0	3,729	23,769	412,996
Tapestry	5.1.0.5	1,502	9,141	97,206

In our evaluation, we differentiate *small* and *large* methods. By small methods we refer to the ones that are in the first, second, and third quartiles of the method's size distribution, measured in lines of code. The rationale behind this division is based on studies [4, 6, 23] that demonstrated that the LOC distribution is usually highly skewed, meaning that in most systems we have many small methods (in the first, second, and third quartiles, according to our convention) and some large methods (in the last quartile). We also analyzed the quartile functions for the MLOC metric in our target systems. As illustrated in Figure 5, our analysis reveals that the distributions of all systems present a heavy-tail behavior. Therefore, we argue that these two categories is the correct design decision for this study.

Another issue that motivates us to evaluate small and large methods separately is because JMove does not provide recommendations for methods with less than four dependencies. In this way, we evaluate JMove both in a friendly context (large methods) and in an unfriendly one (small methods). Table 3 shows the thresholds used to classify small methods in each system—which ranges from

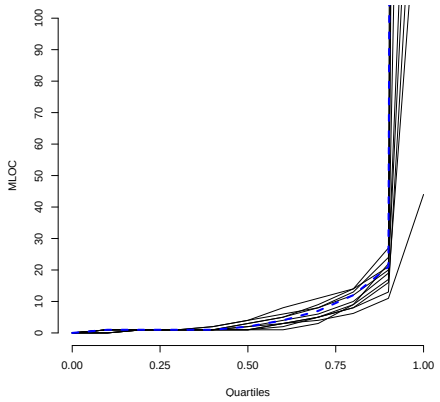


Figure 5: Heavy-tail distributions for the MLOC metric of the target systems

5 LOC (JUnit and MvnForum) to 12 LOC (JFreeChart)—, and the number of small and large methods for each system according to our division criterion.

Table 3: Thresholds used to define small methods

System	**≤ LOC**	**# of small methods**	**# of large methods**
Ant	6	10,210	3,150
Derby	11	30,364	9,937
DrJava	6	7,799	2,565
JFreeChart	12	8,990	2,820
JGroups	10	6,773	2,018
JTopen	8	17,753	5,742
JUnit	5	782	248
Lucene	11	20,723	6,236
MvnForum	5	6,804	2,212
Tapestry	6	7,741	2,472

4.2.2 Gold Sets

To evaluate precision and recall, it is crucial to identify the methods implemented in incorrect classes, which we refer as the gold sets [7]. Typically, the task of generating such sets requires the participation of expert developers in order to manually analyze and classify each method. However, in the context of open-source systems, it is not straightforward to establish a contact with key project developers. For this reason, we manually synthesized a version of each system with well-known methods implemented in incorrect classes, i.e., for each system we moved a few random methods to other random classes.

Figure 6 illustrates an example. The original system (on the left) has three classes A, B, and C: A has methods $a_1, \ldots, a_i$; B has methods $b_1, \ldots, b_j$; and C has methods $c_1, \ldots, c_k$. To generate a gold set with methods implemented in incorrect classes, we changed the location of three methods in this system: method $b3$ (which is incorrectly implemented in class A and that should be moved to its original class B); method $c5$ (which is now incorrectly implemented in B instead of C); and method $a1$ (incorrectly located in C and not in its original class A).

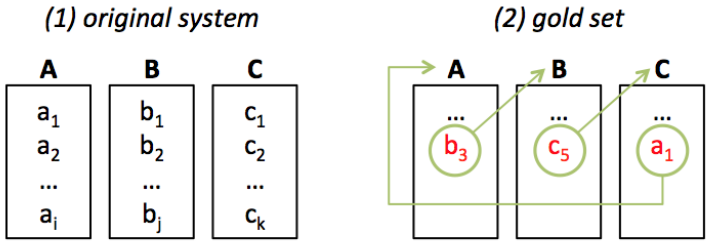


Figure 6: Example of modification performed to generate the gold sets

Specifically in this study, on each system, the third paper's author randomly selects ten small methods and ten large methods to manually move. Suppose he selects to move method m in a class C to a target class C'. In this case, he also checks whether it is possible to move m back from C' to C. This verification is important because otherwise our approach—and also JDeodorant—would never make a recommendation about m in the synthesized system.[1] When a selected method and target class do not attend this condition, a new candidate is randomly generated, until reaching the limit of ten small and ten large methods.

We synthesize for each system S a modified system S' with a *GoldSet* of small and large methods with a high probability of being implemented in incorrect classes for the reasons described next:

- In the case of JUnit, it is reasonable to consider that *methods are likely to be in their correct classes* since it is developed and maintained by a small number of expert developers. We do not claim, however, this system does not contain feature envy instances, but if do, they are potentially justified by rational design choices.

- For the other nine systems, it is not reasonable to assume that all methods are in the correct classes because they are maintained by developers with different levels of proficiency. However, we argue that it is reasonable to assume that at least *most methods are in the correct classes*.

This procedure only inserts an invalid method in the *GoldSet* when the following two unlikely conditions hold for a randomly selected method m and a randomly selected target class C': (a) m is originally implemented in a wrong class in the original system; and (b) C' is exactly the class where this method should have been implemented. For example, when condition (a) holds, but condition (b) does not hold, we still have a valid method in the *GoldSet* because we are basically moving a method implemented in an incorrect class to another class that is also not correct. In total, we manually moved 98 small methods and 97 large methods.[2]

[1] inCode is not affected by this verification because it does not verify Move Method preconditions.

[2] In MvnForum, we only found 8 small methods and 9 large methods respecting the Eclipse Move Method preconditions; in Ant we found 8 large methods respecting these precondiditions.

4.2.3 Precision and Recall

We perform JMove, JDeodorant, and inCode in the modified systems. Each execution generates a list of recommendations *Rec*, whose elements are triples (m, C, C') expressing a suggestion to move m from C to C'. A recommendation (m, C, C') is a true recommendation when it matches a method in the *GoldSet*. Particularly, for a list of recommendations *Rec* generated by JMove or JDeodorant, the set of true recommendations is defined as:

$$TrueRec = \{\ (m, C, C') \in Rec \mid \exists (m, C, C') \in GoldSet\ \}$$

The suggestions from JMove and JDeodorant contain a single destination class, whereas the suggestions from inCode are in the format (m, C, T), where T is a list of possible target classes. In other words, inCode can suggest moving a method to several classes in a single recommendation. For this reason, a suggestion (m, C, T) provided by inCode is correct when there is at least a class $C' \in T$ matching a correct target class, according to the *GoldSet*.

We measure precision only for JUnit, as follows:

$$Precision = \frac{\mid TrueRec \mid}{\mid Rec \mid}$$

We calculate a first recall measure as the ratio of true recommendations by the number of methods in the *GoldSet*, as follows:

$$Recall_1 = \frac{\mid TrueRec \mid}{\mid GoldSet \mid}$$

We also calculate a second recall, defined as:

$$Recall_2 = \frac{\mid \{\ (m, C, *) \in Rec \mid \exists (m, C, *) \in GoldSet\ \} \mid}{\mid GoldSet \mid}$$

This second definition considers the recommendations suggesting moving m to any other class (denoted by a $*$), which is not necessarily the correct one. Thus, $Recall_1 \leq Recall_2$.

4.3 Results

RQ #1 (Precision): Table 4 shows the precision results for JUnit, computed separately for small and large methods. For small methods, JMove is the system with the best precision (21%), followed by JDeodorant (5%), and inCode (0%). We acknowledge the fact of JMove has achieved the best precision results is somehow expected since it never provides recommendations for methods with less than four dependencies. For large methods, the results are 32%, 15%, and 0% for JMove, JDeodorant, and inCode, respectively.

JDeodorant produces more recommendations than JMove. For large methods, JMove and JDeodorant provide 19 and 40 recommendations, respectively. Despite providing twice more rec-

Table 4: Precision Results (%)

System	Small Methods			Large Methods		
	JMove	**JDeo**	**inCode**	**JMove**	**JDeo**	**inCode**
JUnit	21 (3/14)	5 (2/39)	0 (0/0)	32 (6/19)	15 (6/40)	0 (0/0)

ommendations, JDeodorant only matches six large methods in the gold sets, which is the same number methods matched by JMove. For small methods, similar behavior and results are observed. Regarding inCode, there is no recommendation for small and large methods.

Summary: JMove precision ranges from 21% (small methods) to 32% (large methods). JDeodorant produces many recommendations, with a precision of 5% (small methods) and 15% (large methods). inCode produces no recommendations.

RQ #2 (Recall): Table 5 shows the Recall-1 results for small and large methods. In a recommendation, Recall-1 requires both the recommended method and the target class to be correct. For small methods, the median Recall-1 results are 21%, 40%, and 0% for JMove, JDeodorant, and inCode, respectively. Again, it is important to consider that JMove only provides recommendations for methods with at least four dependencies. As a result, methods with less than four dependencies are immediately discarded when evaluating JMove. In fact, 79 out of 98 small methods (81%) have less than four dependencies. Considering this fact, the overall recall achieved by JMove (17%) is close to the maximal result possible with this tool. For large methods, the median Recall-1 results are 60%, 35%, and 10% for JMove, JDeodorant, and inCode, respectively.

Table 5: Recall-1 Results (%)

System	Small Methods			Large Methods		
	JMove	**JDeo**	**inCode**	**JMove**	**JDeo**	**inCode**
Ant	0	60	0	38	25	0
Derby	30	20	20	80	30	0
DrJava	30	70	0	80	50	20
JFreeChart	0	50	0	70	40	10
JGroups	30	40	0	60	20	20
JTopen	0	40	0	60	80	30
JUnit	30	30	0	60	60	0
Lucene	10	20	0	60	30	20
MvnForum	13	75	0	78	44	0
Tapestry	30	0	0	60	20	10
Overall	17	40	2	65	40	11
Average	17	41	2	65	40	11
Median	21	40	0	60	35	10
Std Dev	14	24	6	13	19	11

Table 6 shows the Recall-2 results, which only require the recommended method to be correct. For small methods, the median Recall-2 results are the same as the ones computed for median Recall-1. For large methods, the median results are 79%, 37%, and 10% for JMove, JDeodorant, and inCode, respectively. Therefore, JMove's result improves 32%, when compared to the Recall-1 results.

Table 6: Recall-2 Results (%)

System	**Small Methods**			**Large Methods**		
	JMove	**JDeo**	**inCode**	**JMove**	**JDeo**	**inCode**
Ant	0	60	0	50	25	0
Derby	30	20	20	80	30	10
DrJava	30	70	0	80	50	30
JFreeChart	0	50	0	80	50	10
JGroups	30	40	0	70	30	20
JTopen	0	40	10	60	80	30
JUnit	30	30	0	80	70	0
Lucene	10	20	0	60	30	30
MvnForum	13	75	0	78	44	0
Tapestry	30	0	0	80	20	10
Overall	17	40	3	72	43	14
Average	17	41	3	72	43	14
Median	21	40	0	79	37	10
Std Dev	14	24	7	11	20	13

Summary: Recall results reflect JMove design decision on addressing methods with at least four dependencies. The tool's median recall increases almost three times from small methods (21%) to large ones (60%). It also raises to 79% when identifying only the recommended method (but not the target class) is needed. JDeodorant median recall is 40% for small and 35% for large methods. inCode recall is very low, at most 10% (large methods).

RQ #3 (Performance): To measure runtime performance, we use an Intel Core2 Duo CPU E8400 @3.00GHz with 16GB RAM, with LinuxMint 14 (Nadia), and Java SE Runtime Environment 1.6.0_45. Table 7 reports the time required by the tools to provide their recommendations. On average, JMove requires more than three hours to provide recommendations, while JDeodorant and inCode require 48 and 2 minutes, respectively. Although inCode is faster, it is worth noting that the tool does not check the Move Method preconditions.

To explain JMove's performance, we need to remind that it creates dependency sets for every method in the system and also computes the similarity between such methods and all classes. Moreover, JMove depends on the Eclipse API to check the Move Method refactoring preconditions. Actually, our current performance bottleneck is this checking procedure, which demands more than half of the total execution time. By contrast, JDeodorant implements its own precondition

Table 7: Execution time (hh:mm)

System	JMove	JDeodorant	inCode
Ant	1:08	0:21	0:01
Derby	12:47	3:27	0:04
DrJava	4:39	0:19	0:03
JFreeChart	0:59	0:08	0:01
JGroups	0:44	0:12	0:01
JTopen	4:54	1:39	0:02
JUnit	0:02	0:01	0:01
MvnForum	0:10	0:06	0:01
Lucene	8:52	1:42	0:05
Tapestry	0:18	0:07	0:01
Average	3:27	0:48	0:02
Median	1:03	0:15	0:01

checking functions, which are more efficient than Eclipse's functions. Therefore, we plan to replace Eclipse functions by a native implementation. Last but not least, we claim that our performance is not a major issue; JMove is a prototype tool with no optimizations at all, whereas JDeodorant and inCode are stable tools.

Summary: JMove is four times slower than JDeodorant. However, JMove is still a prototype tool, without any optimization. inCode is the fastest tool, usually running in few minutes.

RQ #4 (Similarity Coefficients): As described in Section 3.2, JMove relies on Jaccard coefficient to estimate the similarity of two dependency sets. We decide to use Jaccard because it is a well-known coefficient, which is also used in other tools, such as JDeodorant. However, there are other set similarity coefficients reported in the literature and that can be used in JMove. Therefore, in this RQ, we evaluate the use of six other coefficients, as listed in Table 8.

Table 8: Similarity Coefficients

Coefficient (ACRONYM)	Definition
1. Jaccard (JAC)	$a/(a+b+c)$
2. Sorenson (SOR)	$2a/(2a+b+c)$
3. Ochiai (OC)	$a/[(a+b)(a+c)]^{\frac{1}{2}}$
4. PSC (PSC)	$a^2/[(b+a)(c+a)]$
5. Dot-product (DP)	$a/(b+c+2a)$
6. Kulczynski (KUL)	$\frac{1}{2}[a/(a+b)+a/(a+c)]$
7. Sokal and Sneath 2 (SS2)	$a/[a+2(b+c)]$

Table 9 reports JMove's *Recall₂* for large methods, when the tool is configured to use Jaccard (as in the previous RQs) and also the coefficients in Table 8. As can be observed, there are no gains in recall when other coefficients are used. Two coefficients produce the best overall *Recall₂* result (72%) and Jaccard is one of them.

We also perform a statistical test over the data in Table 9. The data from Jaccard deviate from normality using Shapiro-Wilk test (p-value = 0.0055). For this reason, we rely on Wilcoxon Signed-Rank Tests to check the null hypothesis of the means remain essentially unchanged ($p > 0.05$). This conclusion is based on the following results between Jaccard and: (a) Sorenson, p-value = 0.6547; (b) Ochiai, p-value = 0.2334; (c) PSC, p-value = 0.715; (d) Dot-product, p-value = 0.6547; (e) Kulczynski, p-value = 0.0917; and (f) Sokal and Sneath 2, p-value = 1. Therefore, there is no statistical difference on the distribution of the recall results provided by Jaccard and by the considered similarity coefficients.

Summary: There are no gains on using other similarity coefficients in JMove, instead of Jaccard.

Table 9: Comparison of similarity coefficients (Recall_2, %)

System	**JAC**	**SOR**	**OC**	**PSC**	**DP**	**KUL**	**SS2**
Ant	50	50	63	63	50	63	50
Derby	80	80	70	70	80	60	70
DrJava	80	80	80	80	80	80	80
JFreeChart	80	80	60	60	80	60	80
JGroups	70	70	70	70	70	60	70
JTopen	60	60	60	60	60	60	60
JUnit	80	70	70	80	70	70	80
Lucene	60	60	50	60	60	50	70
MvnForum	78	78	78	78	78	67	78
Tapestry	80	80	70	80	80	70	80
Overall	72	71	67	70	71	64	72
Average	72	71	67	70	71	64	72
Median	79	74	70	70	74	61	74
Std Dev	11	11	9	9	11	8	10

4.4 Threats to Validity

The threats to validity of this first study are as follows:

External Validity: First, this first study is solely based on randomly applied artificial moves, which may not be representative of a real population of Move Method refactoring opportunities. Second, our subjects are nine open-source systems, randomly selected among the systems in the Qualitas Corpus (plus JUnit). Therefore, we cannot argue that this sample represents the whole population of Java systems, for example. To address these threats, we perform a study with two industrial-strength systems, and judgment by expert developers, as described in Sections 5 and 6.

Internal Validity: First, we acknowledge that the considered gold sets can include false Move Method refactoring opportunities in rare circumstances, as discussed in Section 4.2.2. However, these entries would equally affect the results of the evaluated tools, which at least makes the comparison fair. Second, the gold sets are partitioned in small and large methods. We use the quartiles of the method's size distribution to divide the methods in these two categories. Because this distribution is usually highly skewed, we consider that small methods are the ones in the first three quartiles and large methods are placed in the last quartile. Although one could argue that these two classifications are not sufficient, our analysis revealed that the distributions of our target systems indeed present a heavy-tail behavior.

Construct Validity: When measuring precision, we assume that all methods in JUnit are implemented in their correct classes, which may not be completely true even in systems with a good reputation in terms of design quality. When measuring recall, we compute only the recommendations matching the methods in the gold sets. We never inspected the other recommendations to check, for example, if they indeed represent real Move Method refactoring opportunities in the considered systems. Despite these threats, we claim the reported precision and recall measures provide at least a reliable approximation of the accuracy of the evaluated tools in our synthesized scenario of study. Specifically to mitigate the threat related to the precision measures, we evaluate precision in two other systems, as described in Section 6.

5 Overlapping Study

This section reports a study that compares the overlapping of the recommendations provided by JMove and by JDeodorant, inCode, and Methodbook. The study was designed apart from the first one (Section 4) because Methodbook does not have a publicly available implementation. Therefore, it is not possible to execute Methodbook on the set of open-source systems considered in the previous study. However, the recommendations provided by Methodbook when executed over two systems (JEdit and JFreeChart) are available in the authors' site. Therefore, in this second study we execute JMove, JDeodorant, and inCode over JEdit and JFreeChart and collect the recommendations provided for each tool. We then report the overlapping between the recommendations provided by JMove and by the other recommenders.

5.1 Research Questions

This study aims to answer the following research question:

- ***RQ #1*** – What is the overlap of the recommendations provided by JMove and by the other tools?
- ***RQ #2*** – What is the overlap of the *correct* recommendations provided by Methodbook and the recommendations provided by the other tools?

5.2 Study Design

The study relies on two systems: JEdit (version 3.0), an open-source text editor that supports several programming languages, and JFreeChart (version 0.9.6), an open-source Java API for creating charts. Table 10 presents data on the size of these systems.

Table 10: Target Systems

System	**Version**	**NOC**	**NOM**	**LOC**
JEdit	3.0	434	2,617	47,623
JFreeChart	0.9.6	459	3,271	44,960

For each system, we retrieve the following data from a previous study on Methodbook [3]: (a) a list of *all* Move Method recommendations provided by Methodbook; (b) a list of the *correct* Move Method recommendations provided by Methodbook, according to 56 and 70 developers that evaluated the recommendations for JEdit and JFreeChart, respectively. We also execute JMove, JDeodorant, and inCode over each system and collect the recommendations provided by these tools. Table 11 shows the number of recommendations considered in this study.

Table 11: Number of Recommendations

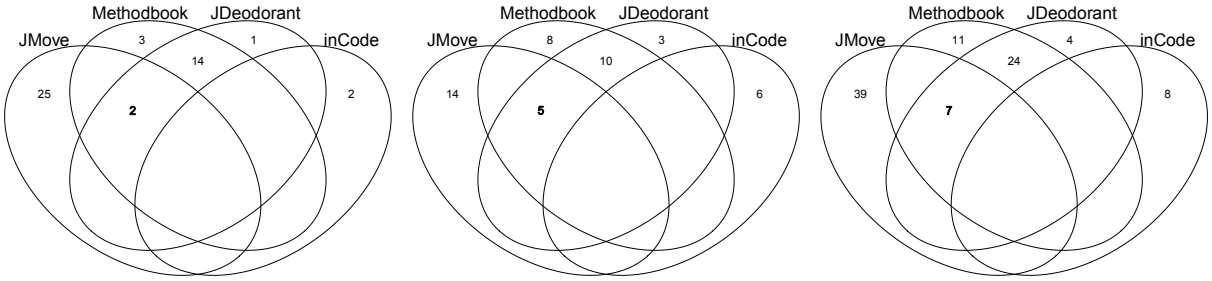
System	**Methodbook**		**JMove**	**JDeodorant**	**inCode**
	All	*Correct*			
JEdit	19	10	27	17	2
JFreeChart	23	10	19	18	6

5.3 Results

In this section, we provide answers for the proposed research questions.

RQ #1 (Overlapping of Recommendations): Figure 7 shows the overlapping of the recommendations provided by JMove and by Methodbook, JDeodorant, and inCode. First, we can observe that there is no overlapping between JMove and inCode. Second, the overlapping between JMove, JDeodorant, and Methodbook is small and includes exactly the same recommendations. For JEdit, this overlapping contains two recommendations and for JFreeChart it contains five recommendations (in bold, in Figures 7(a), 7(b), and 7(c)).

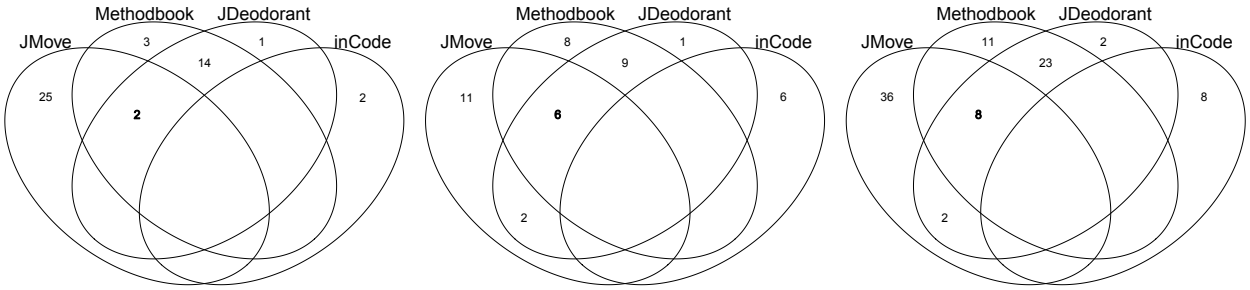
Figure 8 shows a similar overlapping, but considering that two recommendations are identical if they recommend to move the same method (even if the suggested target classes are not the same). The results are similar to the ones reported in Figure 7; the most important difference is that in JFreeChart the overlapping between JMove, Methodbook, and JDeodorant increases from five to



(a) JEdit (b) JFreeChart (c) Total

Figure 7: Overlapping results (two recommendations are considered identical if they recommend to move the same method to the same target class)

six recommendations.



(a) JEdit (b) JFreeChart (c) Total

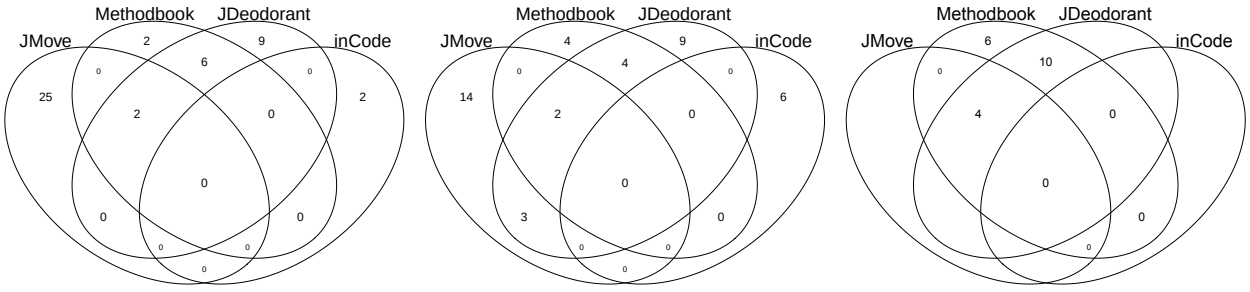
Figure 8: Overlapping results (two recommendations are considered identical if they recommend to move the same method)

RQ #2 (Overlapping of Correct Recommendations): Figure 9 shows the overlapping of the recommendations provided by JMove, JDeodorant, inCode, and the *correct* ones reported by Methodbook. For JEdit, we can observe that JMove found two out of ten correct recommendations produced by Methodbook (and JDeodorant has found eight). For JFreeChart, JMove has also found two out of ten correct recommendations produced by Methodbook (and JDeodorant has found six). These results are discussed in the next section.

5.4 Discussion

For each target system, we inspect the *correct* recommendations provided both by Methodbook and by JDeodorant and that are not provided by JMove. As can be observed in Figure 9, six and four recommendations matched this criterion in JEdit and JFreeChart, respectively.

For JFreeChart, the four recommendations missed by JMove have less than four dependencies and therefore they are not considered by this tool. For example, in Code 5 both Methodbook and



(a) JEdit (b) JFreeChart (c) Total

Figure 9: Overlapping considering only correct recommendations produced by Methodbook

JDeodorant suggested to move the `getX` method from `TimeSeriesCollection` (its current class) to `RegularTimePeriod` and the surveyed developers ranked this recommendation as correct.

```
 1: private long getX(RegularTimePeriod period) {
 2:     long result = 0L;
 3:     switch (position) {
 4:         case (START) : result = period.getFirstMillisecond(workingCalendar); break;
 5:         case (MIDDLE) : result = period.getMiddleMillisecond(workingCalendar); break;
 6:         case (END) : result = period.getLastMillisecond(workingCalendar); break;
 7:         default: result = period.getMiddleMillisecond(workingCalendar);
 8:     }
 9:     return result;
10:}
```

Code 5: Example of method from JFreeChart that JDeodorant and Methodbook recommended to move (but JMove did not)

For JEdit, three of the six recommendations missed by JMove also have less than four dependencies. For example, in Code 6 both Methodbook and JDeodorant suggested to move `readMarkers` from `BufferIORequest` (its current class) to `Buffer`.

Finally, it is important to reinforce that JMove is totally centered on static dependencies; however, it is possible to argue that a static dependency is indeed a means to connect classes which are conceptually related. In other words, indirectly JMove may be recommending classes which share conceptual information, as happens with approaches based on Relational Topic Modeling, as MethodBook. However, in this section, when we compared JMove with MethodBook, we usually found just a small overlap between the recommendations provided by these tools.

Summary: Seven out of ten correct recommendations missed by JMove (but recommended by JDeodorant and Methodbook) are in small methods, with less than four dependencies, which are not considered by JMove. This result reinforces the findings of the first study, when we observed that JMove tends to provide its best results for large methods.

```
private void readMarkers(Buffer buffer, InputStream in) throws IOException {
    buffer.removeAllMarkers();
    StringBuffer buf = new StringBuffer();
    int c;
    boolean eof = false;
    String name = null; int start = -1; int end = -1;
    for (;;) {
        if (eof) {
            break;
        }
        switch(c = in.read()) {
            case -1: eof = true;
            case ';': case '\n': case '\r':
                if (buf.length() == 0) continue;
                String str = buf.toString();
                buf.setLength(0);
                if (name == null) {
                    name = str;
                } else if (start == -1) {
                    try { start = Integer.parseInt(str); }
                    catch(NumberFormatException nf) { start = 0; }
                } else if (end == -1) {
                    try { end = Integer.parseInt(str); }
                    catch(NumberFormatException nf)     { end = 0; }
                    buffer.addMarker(name,start,end);
                    name = null; start = -1; end = -1;
                }
                break;
            default:
                buf.append((char)c);
                break;
        }
    }
    in.close();
}
```

Code 6: Example of method from JEdit that JDeodorant and Methodbook recommended to move (but JMove did not)

6 Evaluation with Experts

This section is divided into five parts. Section 6.1 states the research questions we aim to answer. Section 6.2 presents two industrial-strength systems and the methodology we follow to answer the proposed questions. Sections 6.3 and 6.4 report and discuss the results for each system, respectively. Finally, Section 6.5 lists threats to validity.

6.1 Research Questions

This third study aims to answer the following research questions:

- ***RQ #1***– How does JMove compare with JDeodorant and inCode in real instances of methods implemented in incorrect classes?
- ***RQ #2***– How relevant are JMove's recommendations from the developers' point of view?

6.2 Study Design

In this section, we present the systems we evaluate and the methodology we follow.

Target systems: This study relies on two systems: (i) SGA, a proprietary EJB-based information system used by a major Brazilian university[3], and (ii) Geplanes, an open-source web-based strategic management system.[4] Table 12 presents data on the size of these systems.

Table 12: Systems evaluated by experts

System	NOC	NOM	LOC
SGA	1,056	11,556	27,045
Geplanes	340	3,101	29,046

We select these systems for two main reasons: (i) both systems are mature projects, facing a continuous process of maintenance and evolution, and (ii) we had access to an expert in each system who evaluated the recommendations provided by JMove, JDeodorant, and inCode.

Methodology: We perform JMove, JDeodorant, and inCode on both systems. As detailed in Section 4.2, JMove and JDeodorant generate a list of recommendations, whose elements are triples (m, C, C') expressing a suggestion to move m from C to C'. On the other hand, a suggestion from inCode is a triple (m, C, T), where T is a list of classes.

After generating the list of recommendations provided by each tool, we invited the experts to evaluate the recommendations using a Likert scale [11]. We asked the experts the following question: *How do you rank this Move Method recommendation?* where the answers could be one of the following: *(1) Strongly not recommended*, *(2) Not recommended*, *(3) Neither recommended nor not recommended*, *(4) Recommended*, or *(5) Strongly recommended*. Although we relied on the most expert developer of each software project and asked them to explain their rationale behind each answer, we acknowledge the potential bias of a single interviewed developer and the limited extrapolation of the results, which is further discussed in Section 6.5.

6.3 SGA Results

In the SGA system, JMove did not trigger any recommendation, whereas JDeodorant and inCode triggered 43 and 50 recommendations, respectively. The expert classified every recommendation provided by JDeodorant and inCode as *strongly not recommended*. Since JMove does not recommend methods with less than four dependencies, it is worth noting that nine out of 43 JDeodorant recommendations and none of 50 inCode recommendations have less than four dependencies.

[3]Due to confidentiality reasons, we are omitting the real name of this system.

[4]http://www.softwarepublico.gov.br

He associated the lack of correct Move Method opportunities in SGA to its strict and well-defined architecture, presented in Figure 10. As can be observed, SGA follows a layered architectural pattern where the *ManagedBean* layer is the bridge between Graphical User Interface (GUI) components and business-related components. The *Service* layer implements the core business processes, *Persistence Layer* is responsible for database operations, and the *BusinessEntity* layer implements entity types (e.g., `Professor`, `Student`, etc.).[5] According to the expert, *BusinessEntity* classes should have only fields and their respective *getters* and *setters*. Nevertheless, all recommendations triggered by JDeodorant and inCode suggest to move other methods to such classes.

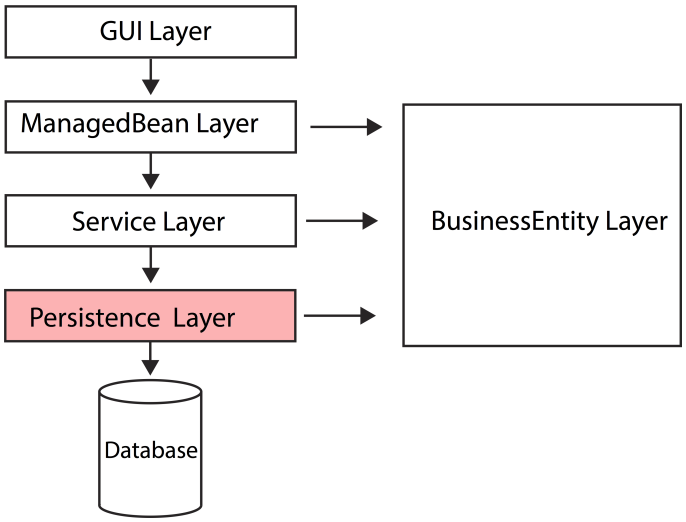


Figure 10: SGA's architecture

JDeodorant and inCode trigger too many *strongly not recommended* refactorings because these tools rely on the number of accesses to members of possible target classes and, by their very nature, entity classes are widely accessed. On the other side, JMove is more robust to such false positives because it does not rely on the number of accesses to other classes. In other words, an access to a class is mapped to a single dependency regardless of the number of times it occurs. In such way, a single dependency—which can represent no more than 25% of the dependency set because we filter out methods with less than four dependencies—is typically not able to attract a method to a class.

Summary: As concluded in the first study (RQ #1, precision), JMove is more robust to false positives than JDeodorant and inCode. In SGA, a proprietary system with probably very few methods implemented in wrong classes, JMove provides no recommendations, while JDeodorant and inCode provide 43 and 50 incorrect recommendations, respectively.

[5]For the sake of comprehensibility, we translated SGA's and Geplanes' identifiers from Portuguese to English.

6.4 Geplanes Results

As reported in Table 13, JMove, JDeodorant, and inCode trigger 41, 194, and 38 recommendations on Geplanes, respectively.

Table 13: Evaluation of the Geplanes' recommendations by an expert

How do you rank this Move Method recommendation?	**JMove**	**JDeodorant**	**inCode**
(5) Strongly recommended	7 (17.1%)	6 (3.1%)	1 (2.65%)
(4) Recommended	3 (7.3%)	6 (3.1%)	1 (2.65%)
(3) Neither recommended nor not recommended	2 (4.9%)	18 (9.3%)	0 (0%)
(2) Not recommended	9 (22%)	28 (14.4%)	5 (13.2%)
(1) Strongly not recommended	20 (48.7%)	136 (70.1%)	31 (81.5%)
Total	**41**	**194**	**38**

JMove: The tool triggers three recommendations classified as *recommended* and seven as *strongly recommended*, i.e., 10 out of 41 recommendations (24.4%) would improve the system quality according to the expert judgment. Regarding the *strongly not recommended* suggestions, 10 out of 20 recommendations are methods having a parameter `WebRequestContext`, which can never be moved according to the planned architecture. In the other 10 recommendations, the current and the suggested classes are structurally very similar. More specifically, these recommendations suggest moving methods between DAO (Data Access Object) classes or between *Service* layer classes. For instance, five recommendations indicate to move methods from class `AnomalyDAO` to class `ManagementUnitDAO`. Although these methods are structurally more similar to the ones in `ManagementUnitDAO`, semantically they should not be moved.

JDeodorant: JDeodorant provides more correct suggestions than JMove and inCode. As reported in Table 13, JDeodorant detects 12 *strongly recommended* or *recommended* refactorings, whereas JMove and inCode detects 10 and 2, respectively. However, JDeodorant provides 194 recommendations, i.e., 373% and 411% more recommendations than JMove and inCode, respectively. As a result, 164 recommendations (84.5%) triggered by JDeodorant are *not recommended* or *strongly not recommended*. Since JMove does not recommend methods with less than four dependencies, it is worth noting that 68 out of these 164 recommendations have less than four dependencies.

inCode: Similarly to the evaluation with open-source systems (Section 4), inCode produces the lowest number of recommendations (38 recommendations) and the worst precision (36 recommendations are *not recommended* or *strongly not recommended*). Once more, it is worth noting that only seven out of the 36 recommendations have less than four dependencies. According to the expert developer, most incorrect recommendations suggest to move methods to entity classes, which should contain only fields and accessor methods.

Summary: JMove has its lowest precision in the Geplanes system (24.4%). Most incorrect recommendations provided by the tool do not respect Geplanes' architecture, e.g., they imply on moving methods from `Facade` to `ServiceLayer` classes. However, JMove outperforms JDeodorant and inCode in Geplanes, since these tools achieve a precision of 6.2% and 5.3%, respectively.

6.5 Threats to Validity

The threats to validity of this study are as follows:

External Validity: Since we evaluate two Java EE applications, one can argue that our results might be in part due to the specific design of these systems. Therefore, we cannot extrapolate our results to other types of systems. However, the underlying design of the evaluated systems is quite different, e.g., Geplanes is mostly based on Servlets/JSP whereas SGA on EJB.

Internal Validity: Our results may reflect personal opinions of the interviewed developers on software architecture and development. Anyway, we interviewed expert developers, with large experience, and who are responsible for the central architectural decisions in their systems. Moreover, to mitigate such a threat, we asked the experts to expose the rationale behind their answers.

7 Conclusion

In this paper, we described a novel approach for recommending Move Method refactorings based on the similarity of dependency sets. We evaluated our approach using a sample of 10 open-source systems with 195 well-defined Move Method refactoring opportunities, including 98 opportunities associated to small methods and 97 opportunities associated to large methods. JMove precision ranged from 21% (small methods) to 32% (large methods) and its median recall ranges from 21% (small methods) to 60% (large methods). In the same scenario, JDeodorant achieves a maximal precision of 15% (large methods) and a median recall of 40% for small and 35% for large methods. As a result, we claim that JMove is indicated to detect Move Method refactoring opportunities involving large methods.

We also compared the recommendations provided by JMove, JDeodorant, Methodbook, and inCode in two open-source systems. A percentual of 70% of the recommendations missed by JMove (but recommended by JDeodorant and Methodbook) are in small methods, with less than four dependencies, which are not considered by JMove. This result reinforces the findings of the first study when we observed that JMove tends to provide its best results for large methods. Moreover, we reported an evaluation with experts in two industrial-strength systems. In one system, we did not provide recommendations, while JDeodorant and inCode triggered 43 and 50 incorrect recommendations, respectively. In the other system, JMove achieved a precision of 24.4%, against 6.2% and 5.3% from JDeodorant and inCode, respectively.

As future work, we intend to integrate JMove with JExtract [32], which is a refactoring recommendation system that aims to reveal Extract Method refactoring opportunities.

JMove and the dataset used in this paper are publicly available at:

https://github.com/aserg-ufmg/jmove

Acknowledgment

Our research is supported by CAPES, FAPEMIG, and CNPq.

References

[1] Nicolas Anquetil and Timothy Lethbridge. Experiments with clustering as a software remodularization method. In *6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255, 1999.

[2] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transaction on Software Engineering and Methodology*, 23(1):1–33, 2014.

[3] 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*, 99:1–10, 2014.

[4] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of Java software. In *21st Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–412, 2006.

[5] Jonathan Chang and David Blei. Hierarchical relational models for document networks. *Annals of Applied Statistics*, pages 124–150, 2010.

[6] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

[7] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, pages 53–95, 2013.

[8] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of feature envy bad smells. In *23rd International Conference on Software Maintenance (ICSM)*, pages 519–520, 2007.

[9] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[10] Martin Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley, 1999.

[11] Susan Jamieson. Likert scales: how to (ab)use them. *Medical Education*, 38(12):1217–1218, December 2004.

[12] Yoshio Kataoka, David Notkin, Michael D. Ernst, and William G. Griswold. Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 736–744, 2001.

[13] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice.* Springer, 2006.

[14] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice.* Springer, 2005.

[15] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *21st International Conference on Software Maintenance (ICSM), Industrial and Tool Volume*, pages 77–80, 2005.

[16] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSM)*, pages 350–359, 2004.

[17] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[18] Petru Florin Mihancea and Radu Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 92–101, 2005.

[19] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[20] Iman Hemati Moghadam and Mel Ó Cinnéide. Code-imp: a tool for automated search-based refactoring. In *4th Workshop on Refactoring Tools (WRT)*, pages 41–44, 2011.

[21] Iman Hemati Moghadam and Mel Ó Cinnéide. Automated refactoring using design differencing. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52, 2012.

[22] Mark Kent O'Keeffe and Mel Ó Cinnéide. Search-based software maintenance. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 249–260, 2006.

[23] Paloma Oliveira, Marco Tulio Valente, and Fernando Lima. Extracting relative thresholds for source code metrics. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 254–263, 2014.

[24] Rocco Oliveto, Malcom Gethers, Gabriele Bavota, Denys Poshyvanyk, and Andrea De Lucia. Identifying method friendships to remove the feature envy bad smell. In *33rd International Conference on Software Engineering (ICSE), NIER track*, pages 820–823, 2011.

[25] William Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[26] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea de Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *28th International Conference on Automated Software Engineering (ASE)*, pages 11–15, 2013.

[27] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology, 2000.

[28] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

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

[30] Max Schäefer and Oege de Moor. Specifying and implementing refactorings. In *25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 286–301, 2010.

[31] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *8th Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1909–1916, 2006.

[32] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending automated Extract Method refactorings. In *22nd International Conference on Program Comprehension (ICPC)*, pages 1–11, 2014.

[33] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *5th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 30–38, 2001.

[34] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[35] Friedrich Steimann and Andreas Thies. From public to private to absent: Refactoring Java programs under constrained accessibility. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443, 2009.

[36] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *17th Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345, 2010.

[37] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4, 2013.

[38] Ricardo Terra and Marco Tulio Valente. Towards a dependency constraint language to manage software architectures. In *2nd European Conference on Software Architecture (ECSA)*, pages 256–263, 2008.

[39] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[40] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342, 2015.

[41] Adrian Trifu and Radu Marinescu. Diagnosing design problems in object oriented systems. In *12th Working Conference on Reverse Engineering (WCRE)*, pages 155–164, 2005.

[42] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, pages 347–367, 2009.

[43] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[44] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, 2012.