Recommending Move Method Refactorings using Dependency Sets

Vitor Sales[†], Ricardo Terra[†‡], Luis Fernando Miranda[†], and Marco Tulio Valente[†]
[†]Federal University of Minas Gerais, Brazil
[‡]Federal University of São João del-Rei, Brazil
{vitormsales,luisfmiranda,mtov}@dcc.ufmg.br, terra@ufsj.edu.br

***Abstract*—Methods implemented in incorrect classes are common bad smells in object-oriented systems, especially in the case of systems maintained and evolved for years. To tackle this design flaw, we propose a novel approach that recommends Move Method refactorings based on the set of static dependencies established by a method. More specifically, our approach compares the similarity of the dependencies established by a source method with the dependencies established by the methods in possible target classes. We evaluated our approach using systems from a compiled version of the Qualitas Corpus. We report that our approach provides an average precision of 60.63% and an average recall of 81.07%. Such results are, respectively, 129% and 49% better than the results achieved by JDeodorant, a well-known move method recommendation system.**

***Index Terms*—Move method refactorings; Recommendation systems; Dependency sets.**

I. 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, creating instances of the Feature Envy bad smell [1]. In fact, there are many studies that rank Feature Envy as the most recurring code smell [2, 3]. 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 in the system's maintainability [4, 5].

Besides the causes and consequences, the refactoring actions to remove a Feature Envy are also well-documented. Basically, a Move Method must be applied to move the method from its current class to the class that it envies [1, 6]. In fact, this refactoring is usually supported by the automatic refactoring tools that are part of most modern IDEs. Therefore, applying the indicated fixing action is not a challenging task in the case of a Feature Envy. On the other hand, before applying this refactoring, maintainers must perform two program comprehension tasks: (a) detect the Feature Envy instances in the source code, and (b) determine the correct classes to receive the methods detected by the first task. 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, the aforementioned tasks are usually performed without the support of any program analysis tools, at least in the case of mainstream IDEs.

To tackle this problem, we propose a novel approach to recommend Move Method refactorings. Basically, our approach detects methods displaying a Feature Envy behavior. In such cases, we also suggest a target class where the detected methods should be moved to. More specifically, our approach is centered on the following assumption: *methods in well-designed classes usually establish dependencies to similar types*. For example, suppose that the class `CustomerDAO` is used to persist customers in a database. Typically, the dependency sets of the methods in this class include common domain types (such as `Customer`) and also common persistence related types (such as `SQLException`). Suppose also that one of such methods, called `getAllCustomers`, was inadvertently implemented in the class `CustomerView`, responsible for user interface concerns. In this case, the dependency set of `getAllCustomers` contains the same domain types as the remaining methods in `CustomerView`, but probably will not contain dependencies to types like `Button`, `Label`, etc, which are common in the methods from `CustomerView`. Therefore, since the dependency set of `getAllCustomers` is more similar to the dependency sets in `CustomerDAO` than to the dependency sets in `CustomerView`, our approach triggers a Move Method recommendation, suggesting to move `getAllCustomers` from the former to the latter.

The paper starts by discussing previous research on strategies for detecting Feature Envy instances (Section II). After that, we detail our recommendation approach, including its main algorithms and functions (Section III). We also present a supporting tool we have implemented, called JMove (Section IV). Since our approach is based on the similarity of dependency sets, we conducted an exploratory study to define the best similarity coefficient to be used (Section V). As a result, we decided to rely on the Sokal and Sneath 2 coefficient [7, 8]. We also evaluate our approach in terms of precision and recall, using a sample of 14 systems with well-defined Feature Envy instances we have synthesized manually (Section VI). We report that our solution provides an average precision of 60.63% and an average recall of 81.07%. Such results are, respectively, 129% and 49% better than the results achieved by JDeodorant [6], a well-known refactoring recommendation system. We also discuss and illustrate with concrete examples the main benefits and limitations of the proposed approach (Section VII). We conclude by presenting our contribution and indicating further research (Section VIII).

978-1-4799-2931-3/13 © 2013 IEEE

WCRE 2013, Koblenz, Germany

II. Related Work

We organized related work in two groups: (a) JDeodorant, which is the system closer to our approach; and (b) other approaches to identify refactoring opportunities, mainly related to methods implemented in incorrect classes.

A. JDeodorant

Proposed by Tsantalis and Chatzigeorgiou, JDeodorant [6, 9] follows a classical heuristic to detect Feature Envy bad smells: a method m envies a class C' when m accesses more services from C' than from its own class. 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. 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. Coupling is defined by the Jaccard distance between all entities in the system (methods and attributes) not belonging to a class and the class itself. On the other hand, cohesion is defined by the Jaccard distance between each entity in the class and the class itself. JDeodorant has been recently extended to also identify Extract Method [10] and Extract Class refactorings [11]. We dedicate an entire subsection to JDeodorant because Section VI compares our approach with this tool.

B. Identification of Refactorings Opportunities

Besides JDeodorant, other approaches have been proposed to help developers and maintainers to identify refactoring opportunities. For instance, Marinescu proposed a set of metrics-based rules to capture deviations from established design principles, including the detection of methods displaying a Feature Envy behavior [12]. The proposed rules allow engineers 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 [13]. However, the task of setting the thresholds—i.e., the acceptable metrics values—is not trivial and is usually guided by empirical studies and past experiences [14]. The *iPlasma* tool and its commercial evolution *InFusion* adopt the proposed strategies [15].

Oliveto et al. proposed an approach, named MethodBook [16], to identify Move Method refactoring opportunities using a technique called Relational Topic Model (RTM) [17]. 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. Therefore, MethodBook would recommend the moving of a given method m to the class C' that contains the largest number of m's "friends".

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 refactorings [18]. However, they do not handle methods located in incorrect classes. Similarly, Seng et al. proposed a search-based approach that uses an evolutionary algorithm to provide a list of refactorings that leads to a behaviorally equivalent system structure with better metric values [19]. Since search-based refactoring approaches usually produce a sequence of refactorings, Harman and Tratt showed how to apply Pareto optimality in multiple sequences of refactorings to achieve a better combination of metrics [20].

Finally, the recommendation approach described in this paper was inspired by our previous work on architecture conformance and repair [21]. More specifically, we proposed a recommendation system, called ArchFix [22], that supports 32 refactoring recommendations to repair 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 located in the wrong class 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 investigate methods that do not necessarily denote an architectural violation.

III. Proposed Approach

Our recommendation approach first 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 m and by the remaining methods in C; and (b) the average similarity between the dependencies established by 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 candidate class to receive m.

In the remainder of this section, we describe the recommendation algorithm proposed in this paper (Subsection III-A), the similarity functions that play a central role in this algorithm (Subsection III-B), and the strategy we propose to decide the most suitable classes to receive a method (Subsection III-C).

A. 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 list T containing the candidate target classes to receive m (line 7). Finally, we search in T for the most suitable class

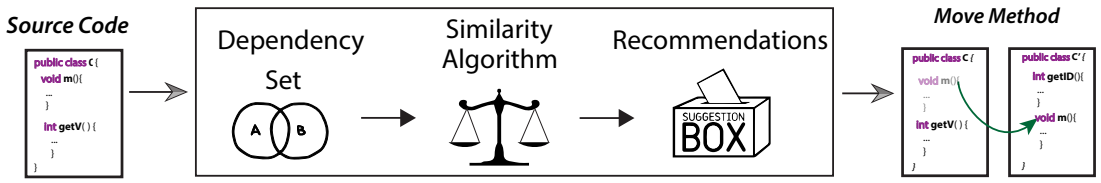

Fig. 1. Proposed approach

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: List with *Move Method* recommendations

```
1:  Recomendations ← ∅
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:     Recomendations = Recomendations + 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 list of candidate classes T to receive m and returns the most suitable one. These functions are described in the following subsections.

B. 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 m and each method m' in C (line 4). In the end, the similarity between m and C is defined as 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 f ∈ 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 `meth_sim(m, m')`, which computes the similarity between the sets of dependencies established by two methods (line 4). Currently, based on an exploratory study where we evaluated 18 similarity coefficients (described in Section V), we are recommending the use of the Sokal and Sneath 2 coefficient [7, 8, 23], defined as:

$$\texttt{meth\_sim}(\texttt{m}, \texttt{m}') = \frac{a}{a + 2(b + c)} \quad (1)$$

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 makes references to. More specifically, we consider the following dependencies:

- *Method calls*: if m calls another method m', the class of m' is added to $Dep(m)$.
- *Field accesses*: if m reads or writes to a field f, the type of f is added to $Dep(m)$. If f is declared in the same class of m, then this class is also added to $Dep(m)$.
- *Object instantiations*: if m creates an object of a type C, then C is included in $Dep(m)$.
- *Local declarations*: if 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 m can raise an exception E or if m handles E internally, then E is added to $Dep(m)$.
- *Annotations*: if m receives an annotation A, then A is included in $Dep(m)$.

When building the dependency sets we ignore the following types: (a) primitive types; (b) types and annotations from `java.lang` and `java.util` (like `String`, `HashTable`, `Object`, and `SupressWarnings`). These types are common to most classes and for that reason they are not relevant when characterizing the dependencies established by methods. They are analogous for example to stop words in natural language processing systems.

Example: Code 1 shows a method `foo` located in a class `Bar`. In this case, we have that

$$Dep(foo) = \{\ Annot, R, B, Z, A, C, D, Bar\ \}$$

Next, we explain why each type was included in this set: `Annot` (annotation used in line 3), `R` (return type in line 4), `B` (parameter declared in line 4 and method call in line 5), `Z` (exception declaration in line 4), `A` (local variable declaration

in line 5 and field access in lines 7 and 9), `C` (method call in line 7), `Bar` and `D` (field access in line 7).

```
1:public class Bar {
2:  private C c;
3:  @Annot
4:  public R foo(B b) throws Z {
5:      A a = b.getA();
6:      if (a != null) {
7:          a.valueR = this.c.getD().valueR;
8:      }
9:      return a.valueR;
10: }
11:}
```

Code 1. Example to illustrate dependency sets

As mentioned, $Dep(m)$ is a set—and not a multiset. Therefore, multiple dependencies to the same type are represented only once. As an example, $Dep(foo)$ has a unique reference to `B`, even though there is a formal parameter of type `B` (line 4) and a method call having a target of type `B` (line 5). In previous work, we provide evidences that traditional sets achieve better precision results than multisets when evaluating the structural similarity of program entities [24].

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

- Methods m that are the only methods in their classes because our approach is based on the similarity between a given method and the remaining methods in its class.
- Methods m whose dependency set has less than four dependencies because few dependencies are more subjected to imprecise or spurious similarity measures. Moreover, by establishing this lower threshold, we also filter out accessor methods (`getters` and `setters`), which by their very nature are rarely implemented in incorrect locations.

C. Target Class Selection

Assume that T is 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 ordered by their similarity with m, as computed by Algorithm 2, the most similar classes first.

To reduce the chances of false positives, a move recommendation is *not* created when: T has less than three classes *and* the difference between the similarity coefficients of C_1 (the first class in the list) and C (the original class of m) is less or equal to 25%. In such cases, we consider that the difference between the dependencies established by C_1 and C are 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 $C_i \in T$ to which the preconditions of such refactoring are attended ($1 \leq i \leq n$). Basically, as usual in the case of refactorings [1, 25], a Move Method has its application conditioned by a set of preconditions, basically 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 as m. When such preconditions are not followed 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 (C, C_{i+1}).

IV. Tool Support

We implemented a prototype tool, called JMove, that supports the approach proposed in this paper. Basically, JMove is an Eclipse plugin that implements the Algorithms 1 and 2, defined in Section III. Figure 2 illustrates the tool's interface, using as example a method called `getAllCustomers` incorrectly implemented in a class designed to support presentation concerns, called `CustomerView`. Basically, JMove extends the IDE interface with a new report panel, used to show the recommended Move Method refactorings.

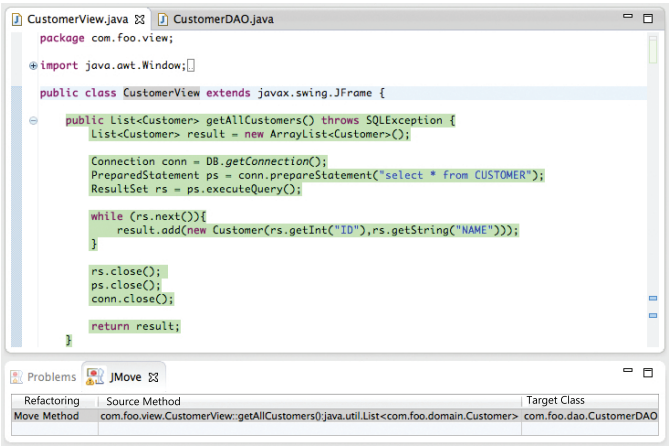

Fig. 2. JMove's interface

Figure 3 presents the three main modules defined by JMove's implementation. The *Dependency Extraction Module* is used to extract the dependency sets that characterize the methods in our approach. For example, the `getAllCustomers` method has the following Dependency Set:

$$Dep(getAllCustomers) = \{Customer, DB, ResultSet, SQLException, PreparedStatement, Connection\}$$

The *Similarity Measurement Module* basically implements the Algorithms 1 and 2. In our running example, this module computes a list T with the following candidate target classes to receive `getAllCustomers`:

$$T = \{[CustomerDAO, 0.94],\ [ProductDAO, 0.86],\ [DepartmentDAO, 0.86],\ \ldots\ \}$$

Finally, module *Recommender System* implements the function `best_class(m, t)` that, given a list T of candidate classes, selects one to receive m, following the criteria defined in Section III-C. In our running example, this function returns the class `CustomerDAO`, since it has many methods similar to `getAllCustomers`. In other words, `getAllCustomers` is more similar, regarding its dependency set, to the methods in `CustomerDAO` than to the other methods in its current class `CustomerView`.

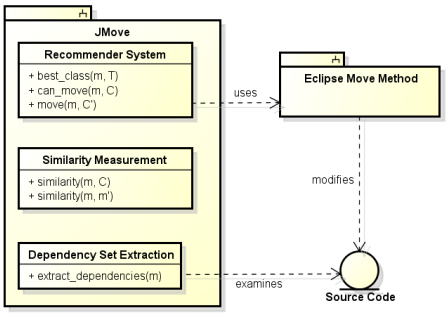

Fig. 3. JMove's architecture

Before making a Move Method recommendation, the *Recommender* verifies whether the recommendation attends the refactoring preconditions. Currently, JMove relies on the preconditions defined by the Move Method automatic refactoring supported by the Eclipse IDE. However, it is well-known that Eclipse implements weak preconditions for some refactorings [26, 27, 28]. For this reason, we strengthened the Eclipse preconditions by supporting the following five preconditions originally proposed by the JDeodorant tool [6].

- The target class should not inherit a method having the same signature with 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 source class fields (including inherited fields).
- The method to be moved should have a one-to-one relationship with the target class.

V. Exploratory Study: Similarity Coefficients

In this section, we report the exploratory study we initially conducted in order to select Sokal and Sneath 2 as the similarity coefficient used by our approach. Basically, in this study, we evaluated the 18 similarity coefficients described in Table I, where a, b, and c are as defined in Section III-B, and d is defined as follows:

$$d = \mid Dep(S) - \ [Dep(m) \ \cup \ Dep(m')] \mid$$

where $Dep(S)$ is a set with the dependencies established by all methods in the system S under analysis .

A. Study Design

We conducted the study using JHotDraw (version 7.6), which is a system commonly used to illustrate object-oriented programming practices and patterns. Its implementation has 80 KLOC, 674 classes, and 6,533 methods. The system has a reputation of having a well-defined and mature design, proposed and implemented by expert developers [29]. For this reason, we based the study on the conjecture that *all methods in JHotDraw are implemented in the correct class*.

We executed Algorithm 1, as described in Section III, multiple times to JHotDraw, considering in each execution a different similarity coefficient. Based on our conjecture, any recommendation should be flagged as a false positive. Therefore, we aim to select the similarity coefficient that generates few recommendations in JHotDraw.

TABLE I
Similarity Coefficients

Coefficient	Definition	Range
1. Jaccard	$a/(a+b+c)$	0–1*
2. Simple matching	$(a+d)/(a+b+c+d)$	0–1*
3. Yule	$(ad-bc)/(ad+bc)$	-1–1*
4. Hamann	$[(a+d)-(b+c)]/[(a+d)+(b+c)]$	-1–1*
5. Sorenson	$2a/(2a+b+c)$	0–1*
6. Rogers and Tanimoto	$(a+d)/[a+2(b+c)+d]$	0–1*
7. Sokal and Sneath	$2(a+d)/[2(a+d)+b+c]$	0–1*
8. Russelll and Rao	$a/(a+b+c+d)$	0–1*
9. Baroni-Urbani and Buser	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+(ad)^{\frac{1}{2}}]$	0–1*
10. Sokal binary distance	$[(b+c)/(a+b+c+d)]^{\frac{1}{2}}$	0*–1
11. Ochiai	$a/[(a+b)(a+c)]^{\frac{1}{2}}$	0–1*
12. Phi	$(ad-bc)/[(a+b)(a+c)(b+d)(c+d)]^{\frac{1}{2}}$	-1–1*
13. PSC	$a^2/[(b+a)(c+a)]$	0–1*
14. Dot-product	$a/(b+c+2a)$	0–1*
15. Kulczynski	$\frac{1}{2}[a/(a+b)+a/(a+c)]$	0–1*
16. Sokal and Sneath 2	$a/[a+2(b+c)]$	0–1*
17. Sokal and Sneath 4	$\frac{1}{4}[a/(a+b)+a/(a+c)+d/(b+d)+d/(c+d)]$	0–1*
18. Relative Matching	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+d+(ad)^{\frac{1}{2}}]$	0–1*

The symbol "$*$" denotes the maximum similarity.

B. Results

Figure 4 presents the number of recommendations generated by each similarity coefficient. For the sake of readability, we do not show the results for the following coefficients that generated more than 100 recommendations: Hamann, Rogers and Tanimoto, Sokal and Sneath, SMC, and Sokal Binary. As reported in Figure 4, the best coefficients were Sokal and Sneath 2 and Russell and Rao, both with 10 recommendations. We decided to use Sokal and Sneath 2 because it is more simple to compute than Russell and Rao, whose computation requires counting the dependencies available in the remaining methods in the system that do not occur in a given pair of methods m and m' (term d in the formula).

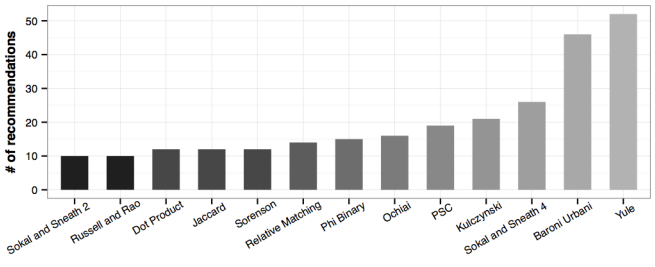

Fig. 4. Move Method recommendations in JHotDraw

C. Threats to Validity

There are two main threats regarding the validity of this initial study. First, as usual, we cannot extrapolate our results to other systems (external validity). However, it is not simple to find systems that confessedly have a mature design, as JHotDraw. Second, even in JHotDraw, it is possible to have methods that are in fact not implemented in the most recommended class (conclusion validity). On the other hand, due to the particular motivation and context behind JHotDraw's implementation, we claim the number of such misplaced methods is really small.

VI. Evaluation

This section reports a study designed to evaluate our recommendation approach, as implemented by the JMove tool.

A. Research Questions

The study aims to answer the following research questions:

- ***RQ #1*** – How does our approach compare with JDeodorant in terms of precision?
- ***RQ #2*** – How does our approach compare with JDeodorant in terms of recall?

Both questions compare our approach with JDeodorant for two reasons: (a) JDeodorant is a well-known solution for identifying Move Method refactoring opportunities; and (b) JDeodorant has a public implementation, that is robust and friendly enough to be used in real-world systems.

B. Study Design

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

Dataset: We evaluate our approach using the systems in the Qualitas.*class* Corpus [30]. This corpus is a compiled version of 111 systems originally included in the Qualitas Corpus [31], but only in a source code format. However, to evaluate tools that depend on the Abstract Syntax Tree (AST) provided by a given IDE, like the one considered in this paper, we need to import and compile the source code. However, this effort is not trivial in the case of systems with many external dependencies. For this reason, we decided to invest in a parallel project aiming the creation of a compiled variant of the Qualitas Corpus.

In fact, our evaluation considers a sample of the systems in Qualitas.*class* Corpus. The criteria to select the systems were as follows: (a) we only considered active projects to avoid outdated systems; (b) we only considered systems having between 500 and 5,000 classes to filter out small and huge systems; (c) we only considered systems whose implementation consists of a single Eclipse project (otherwise we would have to execute the tools multiple times for each project). Among the systems attending our criteria, we selected the first 15 systems, sorted according to their release date from newest to oldest, as stated in the original Qualitas Corpus documentation. In this initial systems selection, JDeodorant did not raise recommendations for two systems: Lucene and iReport. We therefore decided to remove both systems from our sample. Finally, we included JHotDraw to the sample, for the same reasons that motivated its use in our first study (Section V).

Table II presents the final 14 systems considered in the study, including their names, version, number of classes (NOC), number of methods (NOM), and size in terms of lines of code (LOC).

TABLE II
Target Systems

System	Version	NOC	NOM	LOC
Ant	1.8.2	1,474	12,318	127,507
ArgoUML	0.34	1,291	8,077	67,514
Cayenne	3.0.1	2,795	17,070	192,431
DrJava	r5387	788	7,156	89,477
FreeCol	0.10.3	809	7,134	106,412
FreeMind	0.9.0	658	4,885	52,757
JMeter	2.5.1	940	7,990	94,778
JRuby	1.7.3	1,925	18,153	243,984
JTOpen	7.8	1,812	21,630	342,032
Maven	3.0.5	647	4,888	65,685
Megamek	0.35.18	1,775	11369	242,836
WCT	1.5.2	539	5,130	48,191
Weka	3.6.9	1,535	17,851	272,611
JHotDraw	7.6	674	6,533	80,536

Gold Sets: To evaluate precision and recall, it is crucial to identify the methods implemented in the wrong classes, which we refer as the gold sets [32]. Typically, generating such sets would require the participation of expert developers on the target systems 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 the key project developers. For this reason, inspired by the evaluation proposed by Moghadam and Cinnéide in their work on design-level refactoring [33], we manually synthesized a version of each system with well-known methods implemented in incorrect classes, at least with a high probability.

More specifically, we randomly selected a sample including 3% of the classes in each system and manually moved a method of them to new classes, also randomly selected. Before each manual move, we verified the following preconditions: (a) the method selected to be moved must have at least four dependencies in its dependency set because our approach automatically filters out methods not attending this condition (as described in Section III-B); (b) a given class (source or target) can be selected at most once, in order to avoid multiple moves to or from the same class and therefore preserving its current form;[1] and (c) given a source method m in a class C

[1]In practice, multiple moves to or from a same class reduce the chances of returning the moved methods to the original class. For instance, assume a class in which all methods have been moved to another class. In this case, it would be unlikely to recommend the return of the moved methods to the original one, an empty class.

and a target class C', it must be possible to move m to C' and also back to C. This last precondition is important because otherwise our approach—and also JDeodorant—would never make a recommendation about m in the synthesized system. Finally, when a given method and target class do not attend the proposed preconditions, a new candidate is randomly generated, until reaching 3% of the classes in the system.

By following this procedure, we synthesized for each system S a modified system S' with a well-known $GoldSet$ of methods with high probability to be located in the wrong class for the reasons described next:

- In the case of JHotDraw, as claimed in Section V, it is reasonable to consider that *all methods in the original system are in their correct class*, since JHotDraw was developed and maintained by a small number of expert developers. Therefore, we argue that in the modified version of the system, the $GoldSet$ methods are the only ones located in the wrong classes. Due to this special condition, for JHotDraw, we generated five instances of the system with a well-known $GoldSet$.
- In the case of the other systems, it is not reasonable to assume that all methods are in the correct place because such systems are maintained and evolved by different developers. However, we argue that it is reasonable to assume that at least *most methods are in the correct class*. More specifically, the number of methods in such systems range from 4,885 methods (FreeMind) to 21,630 methods (JTOpen). As a consequence, in a sample with this considerable number of methods, the probability of randomly selecting one located in the correct class is much higher than otherwise. Therefore, the $GoldSet$ methods generated according to our methodology have high chances to denote methods in the wrong class.

In fact, the proposed procedure only inserts a invalid method in a synthesized $GoldSet$ 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 a wrong class to another class that is also not correct.

Table III shows the number of methods in the gold sets generated for each system. In total, using the Eclipse support to Move Method refactorings, we moved 475 methods, including 100 methods in five JHotDraw instances.

Calculating Precision and Recall: We executed JMove and JDeodorant in the modified systems. Each execution generated 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 classified as a true recommendation when it matches a method in the $GoldSet$.

TABLE III
GOLD SETS SIZE

System	$\mid GoldSet \mid$	System	$\mid GoldSet \mid$
Ant	25	JRuby	41
ArgoUML	32	JTOpen	39
Cayenne	47	Maven	24
DrJava	18	Megamek	35
FreeCol	17	WCT	29
FreeMind	12	Weka	31
JMeter	25	JHotDraw	20

Particularly, for a list of recommendations Rec generated by JMove or JDeodorant and a given $GoldSet$, the set of true recommendations is defined as:

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

Furthermore, in the case of the Qualitas.*class* systems, we cannot assume that the methods in the gold sets are the only ones implemented in incorrect classes. For this reason, we calculated precision only for the five instances of JHotDraw, as follows:

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

In all systems, we calculated a first recall measure defined as the ratio of the methods covered with the evaluated tools by the number of methods in the $GoldSet$, as follows:

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

We also calculated 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 ratio of methods covered by recommendations suggesting moving m to any other class (denoted by a $*$), not necessarily the correct one. Therefore, we always have $Recall_1 \leq Recall_2$.

C. Results

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

RQ #1 (Precision): Table IV shows the precision results for the five instances of JHotDraw. Our approach—as implemented by the JMove tool—achieved a precision of 60.63% against 26.47% achieved by JDeodorant. In fact, JMove generated less recommendations than JDeodorant (26.2 x 39.2 recommendations in average). Besides that, we generated more true recommendations (15.8 x 10.4 true recommendations in average). In the best case (instance #4), we achieved a precision of 66.67%.

RQ #2 (Recall): Table V shows the $Recall_1$ and $Recall_2$ in each system. Regarding the JHotDraw system, the results

TABLE IV
PRECISION RESULTS FOR JHOTDRAW

System	$\mid Rec \mid$		$\mid TrueRec \mid$		$Precision$ (%)	
	JMove	**JDeo***	**JMove**	**JDeo***	**JMove**	**JDeo***
JHotDraw #1	28	39	16	10	57.14	25.64
JHotDraw #2	27	39	16	10	59.26	25.64
JHotDraw #3	28	38	16	8	57.14	21.05
JHotDraw #4	21	41	14	12	66.67	29.27
JHotDraw #5	27	39	17	12	62.96	30.77
Average	26.20	39.20	15.80	10.40	60.63	26.47
Std Dev	2.95	1.10	1.10	1.67	4.13	3.78
Median	27	39	16	10	59.26	25.64
Max	28	41	17	12	66.67	30.77
Min	21	38	14	8	57.14	21.05

JDeo* stands for **JDeodorant**

TABLE V
RECALL RESULTS

System	$\mathbf{Recall_1}$ (%)		$\mathbf{Recall_2}$ (%)	
	JMove	**JDeodorant**	**JMove**	**JDeodorant**
Ant	84.00	72.00	84.00	84.00
ArgoUML	84.38	56.25	84.38	56.25
Cayenne	72.34	27.66	78.72	38.30
DrJava	83.33	72.22	83.33	72.22
FreeCol	76.47	41.18	76.47	58.82
FreeMind	91.67	58.33	91.67	58.33
JMeter	84.00	60.00	84.00	60.00
JRuby	70.73	58.54	85.37	58.54
JTOpen	89.74	53.85	89.74	58.97
Maven	79.17	45.83	87.50	54,51
Megamek	80.00	51.43	80.00	60.00
WCT	82.76	48.28	86.21	48.28
Weka	77.42	64.52	87.10	74.19
JHotDraw #1-#5	79.00	51.00	79.00	52.00
Average	81.07	54.36	84.11	59.58
Std Dev	5.88	11.83	4.35	11.27
Median	81.38	55.05	84.19	58.68
Max	91.67	72.22	91.67	84.00
Min	70.73	27.66	76.47	38.30

presented in this table are the average of the recall considering the five instances we generated for this system.

Considering $Recall_1$, JMove achieved a recall of 81.07 ± 5.88 (average plus/minus standard deviation) and JDeodorant achieved a result of 54.36 ± 11.83. Therefore, on average, our results are 49.13% better than JDeodorant, in terms of recall. Moreover, our minimal recall was 70.73% (JRuby) and our maximal recall was 91.67% (FreeMind). On the other hand, JDeodorant achieved a maximal recall of 72.22% (DrJava).

Considering $Recall_2$, JMove achieved a recall of 84.11 ± 4.35 and JDeodorant achieved a result of 59.58 ± 11.27. When comparing the results of $Recall_1$ and $Recall_2$, we can observe that $Recall_2$ is 3.74% better than $Recall_1$ in our approach and 9.60% better in JDeodorant. In other words, in both tools the number of full correct recommendations is dominant, i.e., when the tools detect a method in the wrong class, usually they are also able to infer the correct class to receive the method.

Additional Result: Table VI presents the number of recommendations triggered by JMove and JDeodorant for the systems in the Qualitas.*class* Corpus. The results are in the format $tr + r$, where tr are the recommendations that matched a method in the generated gold sets and r are the remaining recommendations, which is not safe to infer whether they represent true recommendations or not. In 11 out of the 13 systems, JDeodorant produced more recommendations than our approach.

TABLE VI
NUMBER OF RECOMMENDATIONS

System	Number of Recommendations	
	JMove	**JDeodorant**
Ant	21 + 118	21 + 156
ArgoUML	27 + 41	18 + 30
Cayenne	37 + 121	18 + 105
DrJava	15 + 81	13 + 293
FreeCol	13 + 162	10 + 281
FreeMind	11 + 44	7 + 60
JMeter	21 + 50	15 + 102
JRuby	35 + 310	24 + 399
JTOpen	35 + 90	23 + 427
Maven	21 + 32	13 + 56
Megamek	28 + 224	21 + 243
WCT	25 + 31	14 + 72
Weka	27 + 175	23 + 327

D. Threats to Validity

Quite similar to the study in Section V, there are three main threats regarding the validity of this second study. First, as usual, we cannot extrapolate our results to other systems (external validity). However, we argue that at least we evaluated a credible sample including 14 real-world systems. Second, we acknowledge that the strategy followed to generate the gold sets can lead to incorrect classifications in rare circumstances, as already discussed in Section VI-B (conclusion validity). However, despite having a low probability of being synthesized, invalid entries in our gold sets affect equally our results and the results generated by JDeodorant, making at least our comparison of the tools fair. Third, we cannot claim that our approach outperforms JDeodorant in a real scenario, since our evaluation is based on artificial moves (conclusion validity). However, we are currently working on a field study, involving real systems and assertions of expert developers.

VII. DISCUSSION

In this section, we discuss our results in qualitative terms. More specifically, we present two examples of Move Method refactorings suggested by our approach for the first modified version of JHotDraw, as described in Section VI.

Example #1: Code 2 shows our first example. The `calculateLayout2` method does not access any service from its current class `AttributeKey`.

Moreover, its dependency set is very different from the dependency sets of the other methods in the class. In fact, we have that:

$$Similarity(calculateLayout2, AttributeKey) = 0.02$$

```
1:Double calculateLayout2(LocatorLayouter locLayouter,
2:  CompositeFigure compositeFigure, Double anchor,
3:  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:         if (bounds == null) {
17:             bounds = r;
18:         } else {
19:             bounds.add(r);
20:         }
21:     }
22: }
23: return (bounds == null) ? new Double() : bounds;
24:}
```

Code 2. First Move Method Example (JHotDraw)

On the other hand, this method is more similar to the methods in the `LocatorLayouter` class. To illustrate, we show next the similarity between `calculateLayout2` and the three methods in `LocatorLayouter`:

$$\begin{aligned} meth\_sim(calculateLayout2, layout) &= 1.00 \\ meth\_sim(calculateLayout2, calculateLayout) &= 0.33 \\ meth\_sim(calculateLayout2, getLocator) &= 0.27 \end{aligned}$$

As result, we have that:

$$Similarity(calculateLayout2, LocatorLayouter) = 0.53$$

For this reason, our approach has correctly recommended to move `calculateLayout2` back to `LocatorLayouter`.

On the other hand, JDeodorant does not make a recommendation to move this method. Basically, in the case of `getter` methods, as the `getLocator` call in line 6, JDeodorant considers that the method envies not the target 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 case.

Example #2: Code 3 shows the second example discussed here. As in the previous example, the `fireAreaInvalidated2` method does not access any service from its current class `DrawingEditorProxy`. However, it calls three methods from `AbstractTool` (lines 2-4). For this reason, JDeodorant infers that `AbstractTool` is a better location for the method. Moreover, it is possible to move the method to this class.

```
1:void fireAreaInvalidated2(AbstractTool abt, Double r){
2:  Point p1 = abt.getView().drawingToView(...);
3:  Point p2 = abt.getView().drawingToView(...);
4:  abt.fireAreaInvalidated(
5:  new Rectangle(p1.x, p1.y, p2.x - p1.x, p2.y - p1.y));
6:}
```

Code 3. Second Move Method Example (JHotDraw)

On the other hand, our approach also suggests to move `fireAreaInvalidated2` back to `AbstractTool`, because the method is more similar to this target class than to its current class, as indicated by the following similarity function calls:

$$\begin{aligned} &Similarity(fireAreaInvalidated2, \\ &\qquad DrawingEditorProxy) = 0.00 \\ &Similarity(fireAreaInvalidated2, AbstractTool) = 0.10 \end{aligned}$$

Limitations: Our approach does not provide recommendations for methods that have less than four dependencies and also for methods that are the single methods in their classes. However, regarding the JHotDraw system, we found only 17 classes (2.5%) having a single method. Among the 6,533 methods in the system, 4,250 methods (65.0%) have less than four dependencies. However, 2,173 of such methods (51.1%) are `getters` or `setters`. We also have 1064 methods (25.0%) that are graphical user interface listeners or utility methods, like `toString`, `equals`, etc. By their very nature, such methods are typically implemented in the correct classes. They are also not considered by other Move Method recommendation systems, including JDeodorant and the search-based approach proposed by Seng et. al [19].

Moreover, we do not recommend moving methods that do not attend the refactoring preconditions because the operation in this case typically requires a more complex restructuring in the source and target classes. Finally, we do not provide suggestions to move fields. However, it is more rare to observe fields declared in the wrong class [6].

Final Remarks: The distinguishing characteristic of our approach is the fact that we depart from the traditional heuristics for detecting move method refactorings. Essentially, such heuristics consider that a method m should be moved from C_1 to C_2 when it access more data from C_2 than from its current class C_1. Instead, we consider that m should be moved when it is more similar to the methods in C_2 than to the methods in C_1. Moreover, we assumed that the dependencies established by a method are good estimators of its *identity*. Therefore, our notion of similarity relies on a similarity coefficient applied over dependency sets, calculated at the level of methods.

Besides checking the traditional heuristic for detecting Feature Envy, JDeodorant only makes a recommendation when the refactoring improves a system-wide metric, that combines cohesion and coupling. Basically, this metric is based on the Jaccard distance between all entities in the system and the original class with the Feature Envy instance. On the other hand, we evaluate the gains of a Move Method refactoring by comparing only the original class of the method with the target class. In fact, recent work has questioned whether high-cohesion/low-coupling—when measured at the level of packages— is able to explain the design decisions behind real remodularizations tasks [34].

VIII. 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 14 systems with 475 well-defined Feature Envy instances.

We achieved an average precision of 60.63% and an average recall of 81.07%. We are currently planning a field study, involving a real system, with possible Feature Envy instances. In this case, we plan to use an expert developer in the system's design to classify our recommendations as true or false positives. In a second study, we also intend to rely on real refactoring, as mined in source code repositories by tools like Ref-Finder [35]. Finally, we plan to compare our solution with other tools, including commercial tools like inFusion [15].

JMove and the dataset used in the paper are publicly available at `http://aserg.labsoft.dcc.ufmg.br/jmove`.

ACKNOWLEDGMENT

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

REFERENCES

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

[2] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *10th International Conference on Quality Software (QSIC)*, 2010, pp. 23–31.

[3] D. I. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, pp. 1–15, 2013.

[4] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.

[5] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 306–315.

[6] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, pp. 347–367, 2009.

[7] R. R. Sokal and P. H. A. Sneath, *Principles of Numerical Taxonomy*. Freeman, 1963.

[8] ——, *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. Freeman, 1973.

[9] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of feature envy bad smells," in *23rd International Conference on Software Maintenance (ICSM)*, 2007, pp. 519–520.

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[11] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

[12] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[13] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *12th Working Conference on Reverse Engineering (WCRE)*, 2005, pp. 155–164.

[14] M. Lanza, R. Marinescu, and S. 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*, 2005, pp. 77–80.

[16] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell," in *33rd International Conference on Software Engineering (ICSE), NIER track*, 2011, pp. 820–823.

[17] J. Chang and D. Blei, "Hierarchical relational models for document networks," *Annals of Applied Statistics*, pp. 124–150, 2010.

[18] M. K. O'Keeffe and M. Ó. Cinnéide, "Search-based software maintenance," in *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 249–260.

[19] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *8th Conference on Genetic and Evolutionary Computation (GECCO)*, 2006, pp. 1909–1916.

[20] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *9th Conference on Genetic and Evolutionary Computation (GECCO)*, 2007, pp. 1106–1113.

[21] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 32, no. 12, pp. 1073–1094, 2009.

[22] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," *Software: Practice and Experience*, pp. 1–36, 2013.

[23] B. S. Everitt, S. Landau, M. Leese, and D. Stahl, *Cluster Analysis*, 5th ed. Wiley, 2011.

[24] R. Terra, J. Brunet, L. Miranda, M. T. Valente, D. Serey, D. Castilho, and R. S. Bigonha, "Measuring the structural similarity between source code entities," in *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2013, pp. 753–758.

[25] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[26] F. Steimann and A. Thies, "From public to private to absent: Refactoring Java programs under constrained accessibility," in *23rd European Conference on Object-Oriented Programming (ECOOP)*, 2009, pp. 419–443.

[27] M. Schäefer and O. de Moor, "Specifying and implementing refactorings," in *25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010, pp. 286–301.

[28] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, 2013.

[29] D. Riehle, "Framework design: A role modeling approach," Ph.D. dissertation, Swiss Federal Institute of Technology, 2000.

[30] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class Corpus: A compiled version of the Qualitas Corpus," *Software Engineering Notes*, pp. 1–4, 2013.

[31] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *17th Asia Pacific Software Engineering Conference (APSEC)*, 2010, pp. 336–345.

[32] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, pp. 53–95, 2013.

[33] I. H. Moghadam and M. Ó. Cinnéide, "Automated refactoring using design differencing," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 43–52.

[34] N. Anquetil and J. Laval, "Legacy software restructuring: Analyzing a concrete case," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 279–286.

[35] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *26th International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.