c-JRefRec: Change-Based Identification of Move Method Refactoring Opportunities

Naoya Ujihara*, Ali Ouni†, Takashi Ishio*, Katsuro Inoue*
*Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
{naoya-u, ishio, inoue}@ist.osaka-u.ac.jp
†Department of Computer Science and Software Engineering, CIT, UAE University, UAE
ouniali@uaeu.ac.ae

***Abstract*—We propose, in this paper, a lightweight refactoring recommendation tool, namely *c-JRefRec*, to identify Move Method refactoring opportunities based on four heuristics using static and semantic program analysis. Our tool aims at identiying refactoring opportunities before a code change is committed to the codebase based on current code changes whenever the developer saves/compiles his code. We evaluate the efficiency of our approach in detecting Feature Envy smells and recommending Move Method refactorings to fix them on three Java open-source systems and 30 code changes. Results show that our approach achieves an average precision of 0.48 and 0.73 of recall and outperforms a state-of-the-art approach namely JDeodorant.**

I. Introduction

Source code of large systems evolves through a process of continuous changes to enhance existing features or add new ones, correct anomalies in design, or fix bugs, etc. [1]. During this process, developers may accidentally or unintentionally implement methods in inappropriate classes, leading to undesirable instances of code smells known as *Feature Envy* [2]. *Feature Envy* is one of the classic and most occurring code smells as pointed out by many studies [3], and thus needs to be prevented/fixed as early as possible.

To fix *Feature Envy* code smell, one of the useful refactorings is *Move Method* [2]. A *Move Method* could be applied to move a method from its original class to the class that it envies. Various refactoring recommendation approaches have been proposed in the literature [4], [5], [6], [7], [8], [9], [10] to help developers to avoid a time-consuming, unrepeatable and non-scalable refactoring process, when performed manually.

However, most existing refactoring approaches aim at recommending refactoring operations that improve quality metrics such as coupling and cohesion, while ignoring the semantic coherence of the program. For example, a refactoring recommendation could move a method `calculateSalary()` from a class `Employee` to a class `Car` because it reduces the overall coupling in the system. However, implemting a method `calculateSalary()` in the class `Car` does not make sense semantically. Furthermore, existing approaches tend to recommend 'global' refactoring solutions to be applied to the entire system [4], [11]. These recommended refactorings often involve classes in the system that are changed rarely during the system's maintenance and evolution and/or classes on which a developer has no or little knowledge. As a consequence, the developer's decision to inspect the recommended refactorings tends to be fastidious, time-consuming and error-prone [4].

To address these issues, we introduce in this paper, a novel refactoring approach to detect *Feature Envy* code smell instances and then identify *Move Method* refactorings to fix them. Our approach is based on currently commited changes and named *c-JRefRec* (change-based refactoring recommendation). Our approach defines a set of heuristics using structural and semantic dependencies to detect refactoring opportunities, through static program analysis.

We evaluate the efficiency of *c-JRefRec* on three non trivial open-source Java systems. Our experiments consist of a random set of 10 commits extracted from the commit log of each studied system. For each commit, we evaluate the efficiency of *c-JRefRec* in terms of precision and recall from a set of known *Feature Envy* instances that are synthesized manually. The obtained results show that our approach is able to detect Feature Envy code smells with an average of 0.48 of precision and 0.73 of recall, and able to identify appropriate Move Method refactorings with an average of 0.42 of precision and 0.68 of recall. We also compared our results against a state-of-the-art technique namely JDeodorant [12], [9].

II. Related Work

A. Definitions

Feature Envy is a classic smell that represents a sign of violating the principle of grouping behavior with related data and occurs when a method is "*more interested in a class other than the one it actually is in*" [2]. More specifically, it is found when a method heavily uses attributes and data from one or more external classes, directly or via accessor operations. Furthermore, in accessing external data, the method is intensively using data from at least one external capsule.

Refactoring, which is defined as "*the process of changing the internal structure of existing code without changing the observable behavior*" [2], is a useful and essential technique for fixing code smells.

B. Identification of Refactoring Opportunities

In recent years, many techniques have been proposed to deal with refactoring recommendation problem, and much efforts have been dedicated to Move Method refactorings.

Tsantalis et al. [12], [9] proposed a refacftoring tool called *JDeodorant* to identify and fix Feature Envy code smells based on coupling and cohesion. Furthermore, *JDeodorant* defines a set of Move Method refactoring preconditions to check whether the recommended refactoring preserve the behavior and the design quality based on entity placement metric. However, the semantic coherence of the refactored program was not considered. Later, Bavota et al. [3] have proposed *MethodBook*, an approach to identify Move Method refactoring opportunities to fix the Feature Envy bad smell. *MethodBook* considers both structural and conceptual relationships between methods to identify sets of methods that share the same responsibilities using Relational Topic Model (RTM).

Furthermore, Sales et al. [8] proposed a Move Method refactoring recommendation approach, namely *JMove* that analyzes a set of static dependencies established by a method. Then it compares the similarity of the dependencies established by a source method with the dependencies established by the methods in possible target classes. Prior to that, Marinescu [13], [14] proposed a set of metrics-based detection rules to identify deviations from established design principles.

Ouni et al. [15], [16] proposed a multi-objective formulation of refactoring to identify refactoring opportunities including Move Method that provide a good trade-off between fixing code smells, and preserving semantic coherence using two heuristics related to vocabulary similarity and structural dependency. Recently, Ouni et al. [4] proposed a search-based refactoring approach with an industrial case study to identify refactoring opportunities, including Move Method, that should provide the best trade-off between improving software quality, fixing code smells and reducing the effort required to apply the recommended refactorings.

III. *c-JRefRec* OVERVIEW

A. Tool Design

Our tool, *c-JRefRec*, takes as input the source code of a program under development in Eclipse IDE, to identify and fix methods implemented in incorrect classes. It employs the AST Parser of Eclipse Java Development Tools (JDT) to analyze the relationships between classes or methods. The tool generates a directed dependency graph $G = (V, E)$ for the entire program where the vertices in V represent methods and fields in the program, and the edges in E represent dependencies (method calls and field access) between them. The tool automatically updates the dependency graph when a developer modifies and saves or compiles a source file. In addition to dependencies, *c-JRefRec* employs a semantic analysis to identify move method refactoring candidates by extracting all code identifiers including names of packages, classes, methods, attributes, and parameters for each class as well as the method to be moved.

Our tool provides two views for Eclipse: *Class State View* and *Refactoring Candidates View* as shown in Figure 1 and 3. The Class State View shows how cohesion and coupling of classes are affected by a code change, by comparing the new dependency graph after change with the original graph. Then, the Refactoring Candidates View shows candidates of Move Method Refactoring, based on the dependency graph and the identifiers extracted from files.

B. Class State View

The view provides four metrics to show cohesion and coupling among classes as follows.

- $methods(C)$ is the number of methods defined in class C, excluding abstract methods. A higher value means that the class has a larger responsibility in a system.
- $edges(C)$ is the total number of incoming edges and outgoing edges connected to members defined in class C. The higher value means that the cohesion of the class is lower.
- $clients(C)$ is the number of classes which use any methods or fields of class C. The higher value means that the cohesion of the class is lower.
- $dependents(C)$ is the number of classes whose method is called or field is accessed by methods in class C. The higher value means that the cohesion of the class is lower.

During a change task, the view lists all modified classes and their client/dependent classes and shows their original values of metrics before the change task and the differences caused by the code change. Since this view is automatically updated when source code is saved/compiled, the developer can know the current number of methods and dependencies added and/or removed in the change. Developers can request a recommendation of refactoring candidates by clicking on a class name in the list as shown in Figure 3.

C. Refactoring Candidates View

This view shows the recommended move method refactoring. Our tool lists possible refactoring candidates, and then evaluates them using three structural heuristics and one semantic heuristic.

The structural heuristics use by the following metrics.

- $\Delta edges(R, C)$ is the number of edges to be added or removed by applying a move method refactoring R.
- $\Delta clients(R, C)$ is the number of client classes to be added/removed by applying a move method refactoring R.
- $\Delta dependents(R, C)$ is the number of dependent classes to be added or removed by applying a move method refactoring R.

The semantic heuristic is represented by a semantic similarity, assuming that a method m should be moved from class c_1 to c_2 if m is more similar to methods in c_2 than methods in c_1. We capture the semantic similarity between a method m and a class c as $SS(m, c) = cosine(tf-idf(m), tf-idf(c))$ using tf-idf vectors where methods and classes are regarded as individual documents.

D. Identification of Move Method Refactoring Candidates

Our tool identifies Move Method Refactoring opportunities, i.e., methods located in inappropriate classes using the following condition: $\Delta edges(R, C_{original}) + \Delta edges(R, C_{target}) + \Delta clients(R, C_{original}) + \Delta clients(R, C_{target}) +$

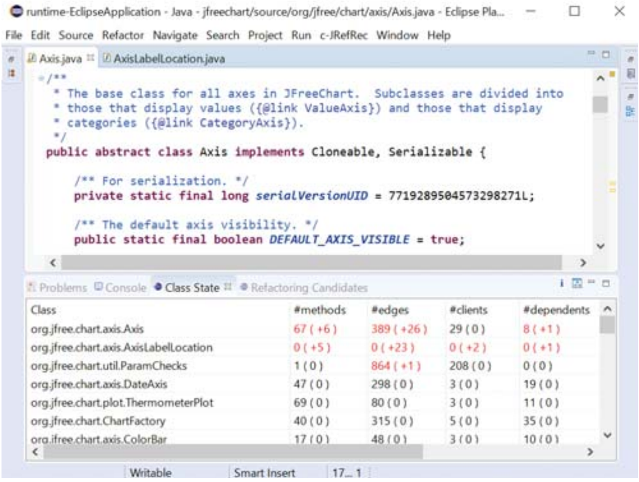


Figure 1: Class State View (before refactoring).

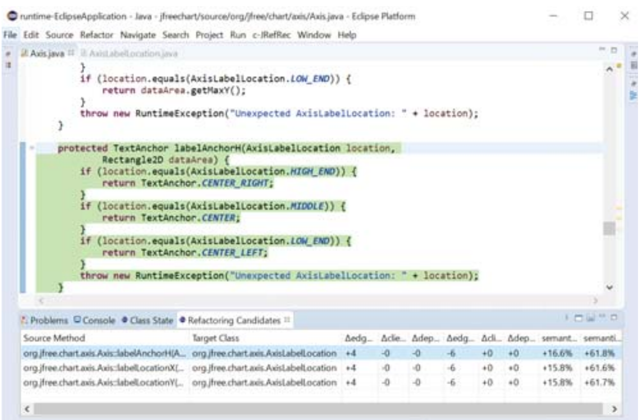


Figure 2: Refactoring Candidates View.

$\Delta dependents(R, C_{original}) + \Delta dependents(R, C_{target}) < 0$ AND $SS(m, C_{original}) < SS(m, C_{target})$

This view shows that not only the name of the method to be moved and the name of target class, but also the current value of each dependency metrics showing if there is an increase/decrease by the move method refactoring. So, a developer can easily decide to apply the refactoring or not.

IV. Illustrative Usage Scenario

We demonstrate the usefulness of *c-JRefRec* in a realistic environment setting. Using our research prototype, we perform a use case scenario in respect to refactoring decisions. A video highlighting the main features of the tool is available at [17].

We chose the popular open source system JFreeChart as a target software to show how *c-JRefRec* works based on the code commit ID 'c7e8c72' recorded on github. In this commit, a new class org.jfree.chart.axis.AxisLabelLocation is created, while some other fields and methods are added/changes in the class org.jfree.chart.axis.Axis.

The Class State View is automatically displayed when a save or compile action is performed. This view is also dynamically updated every time the source code is changed and saved. Figure 1 shows the view after this commit, which shows how many values are changed in this task. More specifically,

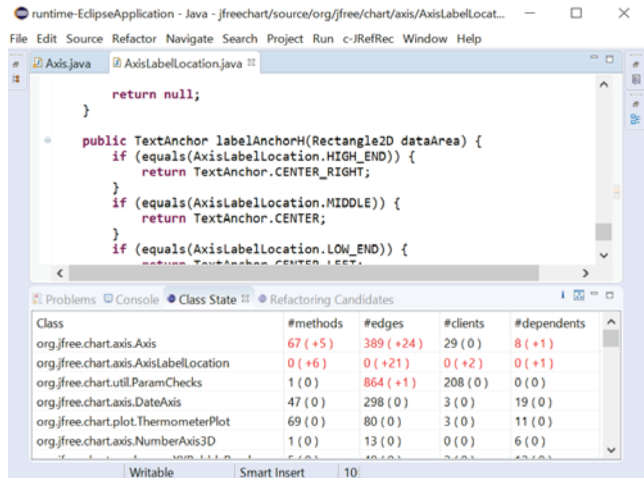


Figure 3: Class State View (after refactoring).

the view displays the class name, the number of methods contained in the class, the number of edges contained in the class, the number of client classes that use the class, and the number of dependent classes it uses. Since AxisLabelLocation is a newly created class, the number of methods is increased by 5 from 0, the number of edges is increased by 23 from 0, the number of client classes is increased by 2 from 0, and the number of dependent classes is increased by 1 from 0. As for the class Axis which is also modified, the number of methods is increased by 6 from 67, the number of edges is increased by 26 from 389, and the number of dependent classes is increased by 1 from 8.

Furthermore, if the developer wants to know if there is a recommended move method refactoring for AxisLabelLocation, then she has to click on that line. As a result, the Refactoring Candidates View is automatically opened as shown in Figure 2. As can be seen in the figure, this view displays the method name to be moved, the name of the target class, as well as different metric values that simulates the refactoring to support the developer taking his decision. These metrics include the number of increased edges in the source class, the number of client and dependent classes to be reduced from the source class, the number of reduced edges and increased client classes from the target class, the number of dependent classes to be increased to the target class, and the semantic similarity between the method and each of the source and target classes. For example, in this commit, the method labelAnchorH() is a newly added method in the class Axis. The tool shows that the number of edges has increased in the source class by 4, but the number of edges to be reduced from the target class is 6. In other words, it can be seen that by performing this refactoring, the number of edges will decrease by 2, thus reducing coupling and increasing cohesion. In this way, the developer can see how to improve the cohesion degree and coupling degree by refactoring not only the method to be moved but also in the target class. Moreover, the semantic similarity between the method and the source class is about 0.166, and the semantic similarity between the method and the target class is about 0.618. The semantic similarity means that

TABLE I: STUDIED SYSTEMS.

System	Version	#classes	#methods	KLOC
JFreeChart	1.0.13	504	7,551	91.174
JMeter	3.0	1,055	8,561	101.501
JUnit	4.4	350	1,254	10.025

the method is more similar to the target class than the source class. These values meet the identification of move method refactoring candidates conditions we defined in sectionIII-D. So, *c-JRefRec* suggests that the labelAnchorH() method should be moved from the Axis class to the AxisLabelLocation class. The developer could apply this refactoring candidate if he considers that it makes sense from a semantic point of view.

Figure 3 shows the Class Status View after this recommended refactoring is applied. When refactoring is applied, the Class State View is automatically updated. As shown in Figure 2, the number of edges has decreased by 2. Finally, by clicking to the recommended refactoring, the labelAnchorH method added in this change task is automatically moved to its envied class AxisLabelLocation.

V. EVALUATION

This section reports the evaluation of *c-JRefRec*. We also compare *c-JRefRec* with a popular existing refactoring tool, JDeodorant [9], [12], which is based on coupling and cohesion improvements for the entire program. Our replication package is available online on [17].

We designed our experiments to address the two following research questions:

- **RQ1.** What is the accuracy of *c-JRefRec* in identifying Feature Envy code smells compared to JDeodorant?
- **RQ2.** What is the accuracy of *c-JRefRec* in recommending Move Method refactorings compared to JDeodorant?

A. Analysis Method

We evaluate our approach on a benchmark of three non-trivial Java open-source systems namely, JFreeChart[1], JMeter[2], and Junit[3], which are summarized in Table I.

To answer our research questions, we need a set of well-known Feature Envy instances and their corresponding refactorings (ground truth). ATo this end, we manually synthesized a gold set of Feature Envy instances and their refactorings. For each studied system, we randomly selected 10 commits from the control version archive. Then, for each commit, we randomly selected two methods and manually moved them to other random classes that are changed in that commit. We used Eclipse to apply these moves for our gold set, so that only move methods that comply with the default Eclipse preconditions are included in the gold set. So that, for each system, we have 20 Feature Envy instances.

To answer **RQ1**, we execute both *c-JRefRec* and Jdeodorant on the dataset. Then we calculate the precision and recall score of each approach in identifying the instance in the Gold set. More specifically, precision and recall are calculated as follows:

$$Detection\_Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Detection\_Recall = \frac{TP}{TP + FN} \tag{2}$$

where TP (*True Positive*) corresponds to a Feature Envy instance identified by the approach and also in gold set; FP (*False Positive*) corresponds to an instance identified by the approach, but not in the gold set; FN (*False Negative*) corresponds to an instance in the gold set, but not identified by the approach.

To answer **RQ2**, we check whether *c-JRefRec* and JDeodorant are able to recommend Move Method refactorings to return back each Feature Envy instance to its original class based on the gold set. We use precision and recall to measure the accuracy of each approach as follows:

$$Refactoring\_Precision = \frac{\#of\ correct\ refactorings}{\#of\ recommended\ refactorings} \tag{3}$$

$$Refactoring\_Recall = \frac{\#of\ correct\ refactorings}{\#of\ refactorings\ in\ the\ gold\ set} \tag{4}$$

B. Results

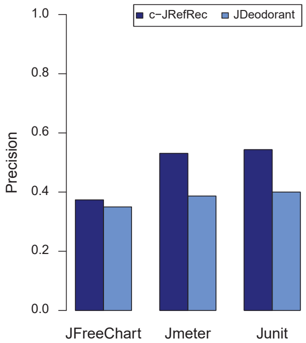
Results for RQ1. Figure 4 present the results for RQ1. We observe that for the 10 commits of each system, *c-JRefRec* achieves an average precision for detecting Feature Envies of 0.48, while an average precision of 0.38 is achieved by JDeodorant for the three systems. Moreover, *c-JRefRec* achieved a maximal precision of 0.54 with JUnit, and JDeodorant achieved also a maximal precision with Junit but with a score of 0.4. In terms of recall, for the three systems and over the 30 commits, we observe that *c-JRefRec* achieves an average recall of 0.73, while JDeodorant achieves an average recall of 0.25. Moreover, *c-JRefRec* achieved its maximal recall of 0.8 with both JMeter and Junit, while JDeodorant achieved its maximal recall with 0.35 for JMeter. Based on these results, both *c-JRefRec* and JDeodorant do not show any particular sensitivity with the size of studied systems.

In more details, we observe that *c-JRefRec* achieved significantly higher recall results (0.73) than precision (0.48) as it tends to identify a relatively larger number of Feature Envy candidates leading to a reduced precision score comparing to recall. Inversely, JDeodorant has lower recall results than precision as it recommends a limited number of smell instances.

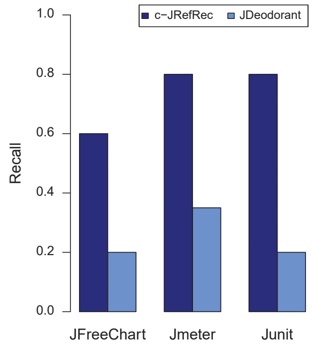
Results for RQ2. Figure 5 report the obtained results for RQ2. For all the 30 commits of the three systems, *c-JRefRec* achieved an average precision of 0.42 and an average recall of 0.68, while JDeodorant achieved 0.38 and 0.25 of precision and recall, respectively. In both tools, the achieved detection results were slightly better than the refactoring results. That is, in both techniques when a method is detected as a Feature Envy, i.e., in an incorrect class, they were also able, in most cases, to identify their correct class that the method originally belonged to.

In most situations where *c-JRefRec* was not able to determine the correct class, we noticed that either there were few connections to the original class (less than 3) or there are other

[1]http://www.jfree.org/jfreechart
[2]http://jmeter.apache.org
[3]http://junit.org/junit4

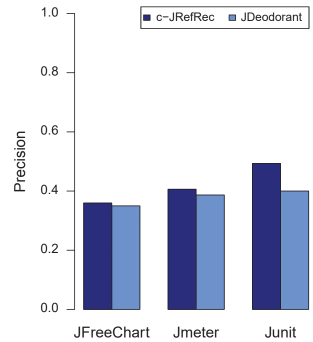


(a) Average precision score for Feature Envy detection.

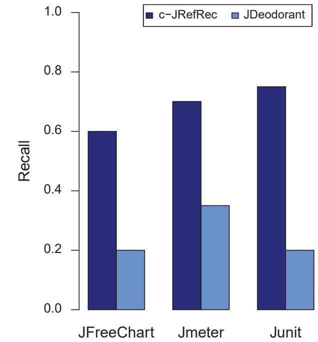


(b) Average recall score for Feature Envy detection.

Figure 4: The average precision and recall results for Feature Envy detection achieved by each of *c-JRefRec* and JDeodorant for the three studied systems.



(a) Average precision score for Move Method.



(b) Average recall score for Move Method.

Figure 5: The average precision and recall results for Move Method refactorings recommendation achieved by each of *c-JRefRec* and JDeodorant for the three studied systems.

classes with the same or higher number of connections, thus preventing the original class from appearing in the first rank.

For the three systems, we observe that *c-JRefRec* achieves better Move Method recommendation results than JDeodorant. Although our approach starts from the traditional heuristics for Move Method, it extends them with the (1) *dependency graph*, (2) the semantic similarity (SS), and (3) considering the importance of both source to target class and vice versa using the structural metrics. Moreover, JDeodorant is based on the idea that a method m should be moved from c_s to c_t if it access more data from c_t than from its original class c_s. However, using the semantic similarity (SS) heuristic, *c-JRefRec* considers that m should be moved if its vocabulary is more similar to the methods in c_t than to the methods in c_t. On the other hand, JDeodorant only makes a recommendation when the refactoring improves metrics for the entire system, based on cohesion and coupling, regardless the current changes performed by a developer.

VI. Conclusion and Future Work

We presented *c-JRefRec*, a novel refactoring recommendation approach that relies on code commits to identify Feature Envy smells and recommend Move Method refactorings to remove them. The proposed approach is based on five heuristics using static and semantic program analysis. We evaluated our approach on three Java open-source systems and 30 commits. Our results show that *c-JRefRec* outperforms a popular state-of-the-art technique and achieves precision and recall scores at 0.48 and 0.73, respectively, in detecting Feature Envy smells, and a precision of 0.42 and recall of 0.68 in recommending correct Move method refactorings. As part of our future work, we plan to compare *c-JRefRec* with other available refactoring tools and conduct an empirical study on different systems with developers to evaluate our approach in real-world scenarios and get more feedback. Moreover, we plan to extend *c-JRefRec* to support more code smells and refactoring operations.

Acknowledgment. This work was supported by JSPS JP25220003, JP26280021, and JP15H02683, by Osaka University Program for International Joint Research, and by the Research Start-up (2) 2016 Grant G00002211 funded by UAE University.

References

[1] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] B. Gabriele, O. Rocco, G. Malcom, P. Denys, and L. Andrea De, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2014.

[4] A. Ouni, M. Kessentini, H. Sahraoui, I. Katsuro, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, 2016.

[5] X. Ge and E. Murphy-Hill, "Benefactor: a flexible refactoring tool for eclipse," in *ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 19–20.

[6] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1106–1113.

[7] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[8] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 232–241.

[9] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[10] A. Ouni, M. Kessentini, H. A. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, pp. 18–39, 2015.

[11] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," *Journal of Systems and Software*, p. to appear, 2016.

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

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

[14] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005, pp. 155–164.

[15] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 347–356.

[16] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[17] "http://sel.ist.osaka-u.ac.jp/people/naoya-u/c-JRefRec."