Finding Extract Method Refactoring Opportunities by Analyzing Development History

Ayaka Imazato, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email:{i-ayaka, higo, k-hotta, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Refactoring is an important technique to improve maintainability of software, and developers often use this technique during a development process. Before now, researchers have proposed some techniques finding refactoring opportunities for developers. Finding refactoring opportunities means identifying locations to be refactored. However, there are no specific criteria for developers to determine where they should refactor because the criteria differ from project to project and from developer to developer. In this study, we propose a technique to find refactoring opportunities in source code by using machine learning techniques. Machine learning techniques enable to flexibly find refactoring opportunities by the characteristics of target projects and developers. Our proposed technique learns information on the features of refactorings conducted in the past. Then, based on this information, it suggests some refactorings on given the source code to developers. We investigated three research questions with five open source projects. As a result, we confirmed that the proposed technique was able to find refactorings with high accuracy.**

***Index Terms*—Refactoring, Extract Method, MSR**

I. Introduction

Identifying refactoring opportunities manually is a laborious task because the amount of source code is usually huge. Besides, developers might overlook the locations to be refactored in manual identification process. Hence, it is necessary to automatically find refactoring opportunities for developers to refactor the source code efficiently. Some research proposed techniques to find refactoring opportunities for developers [1], [2], [3], [4]. On the other hand, there are no specific criteria to determine the locations to be refactored because the refactoring opportunities might differ from project to project and from developer to developer. For example, a refactoring pattern that is frequently conducted in a certain project is not necessarily to be regarded as much as important in other projects. Hence, if we find locations to be refactored based on certain criteria, it is possible that the found locations are beneficial for some users, but not beneficial for other users. That is, it is important to find refactoring opportunities by the characteristics of projects and users to provide information that is beneficial for all users [5].

In this study, we propose a technique to automatically find *Extract Method* refactoring opportunities because *Extract Method* is one of the refactorings that are often conducted [6]. The proposed technique analyzes development histories of software to obtain information about features of methods which existed in the past, and whether *Extract Method* refactoring was applied to them or not. Then, based on this information, the proposed technique constructs a model to identify methods in latest source code to which *Extract Method* should be applied by using machine learning techniques.

II. Extract Method Refactoring

Extract Method is an operation that extracts a part of an existing method as a new different method. Its operations are as follows: first, a developer extracts a part of a target method as a new different method. Then, she/he alters the new method created by the extraction as necessary to maintain external behavior of the software. Finally, she/he replaces the extracted part with a statement to call the new method.

Extract Method is one of the refactoring patterns that are frequently conducted [6]. Furthermore, dividing a multi-functional method by *Extract Method* helps not only to improve maintainability but also to promote reuse of a method when developers need to implement similar functions in the future. Thus, rearrangement of source code by *Extract Method* is very effective for software development.

III. Proposed Technique

Figure 1 shows an overview of the proposed technique. The proposed technique consists of the following two phases.

Phase A (Learning): this phase takes the development history of a target software system as its input. It identifies refactoring conducted in the past and provides a learning model by learning the refactorings. For a given method, the model tells whether it should be refactored or not.

Phase B (Predicting): this phase takes two information as its input: one is the learning model built in Phase A, and the other is source code on which developers want to conduct prediction. This phase collects all the methods in the given source code, applies the given model to them, and provides a list of methods that should be refactored.

The former phase consists of the following four steps.

STEP A-1: we identify all the methods to which *Extract Method* refactoring was applied in the past. Hereafter, we call those methods *refactored methods*.

STEP A-2: we gather methods to which *Extract Method* refactoring was not applied from the same development history. This paper calls these methods *non-refactored methods*. Methods detected in STEP A-1 and STEP A-2 are used as training data.

STEP A-3: we identify all the program elements in each method included in the training data. The proposed technique

0730-3157/17 $31.00 © 2017 IEEE
DOI 10.1109/COMPSAC.2017.129

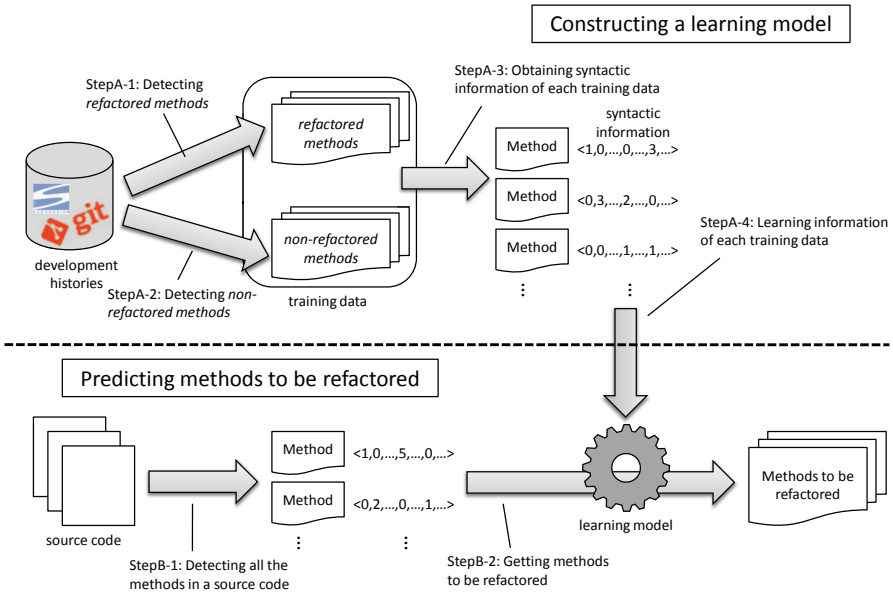


Fig. 1. An Overview of the Proposed Technique

counts the number of appearances for each type of elements in the method. Collected data from each method is converted to a vector. We use the vector as the syntactic information of the method.

STEP A-4: for each method in the training data, we learn its syntactic information and whether *Extract Method* has been applied to it or not. The proposed technique provides a learning model by learning all the methods in the training data. The learning model predicts whether *Extract Method* should be applied to a given method with the syntactic information of it.

The latter phase requires the following two steps.

STEP B-1: we detect all the methods in the given source code when we give the source code that we want to get methods to be refactored in the source code. Then, we obtain their syntactic information.

STEP B-2: we adopt the given learning model constructed to all the methods obtained in STEP B-1, which provides the list of methods that should be refactored.

The rest of this section explains each step of Phase A.

STEP A-1: Detecting Refactored Methods

This step gets assistance of Kenja[1], which is a tool to detect *Extracted Method* refactorings conducted in the past. It takes the development history of a target project managed with version control system Git as its input, and it detects *Extracted Method* refactorings that were conducted during the development process. It reports the following information for each of the *Extract Method* refactorings.

revision: the revision in which the refactoring was conducted.
targetMethod: the method where the refactoring was conducted.
extractedMethod: the method generated by the refactoring.

[1]https://github.com/niyaton/kenja

similarity: the degree of similarity between the part extracted from *targetMethod* and *extractedMethod*, which is calculated as follows.

$$similarity(s_1, s_2) = \frac{|SH(s_1) \cap SH(s_2)|}{|SH(s_1) \cup SH(s_2)|} \quad (1)$$

where, $SH(s)$ refers to the set of all the word pairs in a given document s whose two words of every word pair appear continuously in the document[2] [7]. If a method m include four tokens a, b, c, and d, $s(m)$ includes ab, bc, and cd.

Kenja detects refactorings by analyzing the changes on source code between each of two consecutive revisions. The details of the procedure are described below.

1) Detecting methods that were newly generated by the change as candidates for *extractedMethod*.
2) Detecting methods which were touched by the change as candidates for *targetMethod*.
3) Checking whether *extractedMethod* is called in code fragments that were added to *targetMethod*. The remaining procedure is performed only if *extractMethod* is called.
4) Calculating the *similarity* between *extractedMethod* and code fragments deleted from *targetMethod* with formula (1).
5) Checking whether the *similarity* value exceeds a given threshold.

We used 0.3, which is the default value of Kenja.

STEP A-2: Detecting Non-Refactored Methods

In this step, we obtain *non-refactored methods*. The proposed technique randomly selects a *refactored method* out of the methods detected in STEP A-1. Then, it randomly selects a method to which *Extract Method* was not applied in the same file at the same revision as the *refactored method*. Moreover, we check whether *Extract Method* has never been applied to the selected method during the past development history.

[2]Kenja regards each token as a word.

Algorithm 1 Detecting *non-refactored methods*

Input: $refactoredList$, num
Output: $nonRefactoredList$

```
1: nonRefactoredList ← ∅
2: while num ≠ getSize(nonRefactoredList) do
3:    method ← getRandomElement(refacteredList)
4:    revision ← getRevision(method)
5:    file ← getFileName(method)
6:    candidate ← getRandomMethod(revision, file)
7:    if !isInList(nonRefactoredList, canadidate) then
8:       if !isInList(refactoredList, candidate) then
9:          add(nonRefactoredList, candidate)
10:      end if
11:   end if
12: end while
13: return nonRefactoredList
```

We do not regard the method as a *non-refactored method* if *Extract Method* was applied to the method of certain revision even once. We repeat the above operations until collecting the number of methods required to construct a learning model. Furthermore, when collecting methods in this way, there is a possibility that the same method could be selected more than once as a *non-refactored method*. To prevent such multiple selections, the proposed technique does not add an obtained *non-refactored method* to the training data if the method already exists in the training data.

Moreover, the *non-refactored methods* should not be *methods that were not refactored* but *methods such that developers judged they do not require to be refactored*. The rationale behind this is that a method which was not refactored actually might have required refactoring but have not been refactored due to some reasons, including overlooking or time pressure. Including such a method in the training data will decrease the accuracy of prediction. On the other hand, methods in the same file as *refactored method* at the same revision are more likely to be considered whether to be refactored than methods in other files. Hence, we obtain methods only in the same file and at the same revision as the method to which *Extract Method* was applied in order to collect only methods that were not refactored intentionally.

Algorithm 1 shows the algorithm to detect *non-refactored methods*. The input variables are as follows.

refactoredList: a list of *refactored methods* (methods detected in STEP A-1)
num: the number of *non-refactored methods* that we need as training data

The output variable is as follow.

nonRefactoredList: a list of *non-refactored methods*

Besides, functions in Algorithm 1 are as follows.

getSize: returns the number of elements included in a given list
getRandomElement: returns an element that is randomly selected from the given list
getRevision: returns a revision where the given *refactored method* was refactored
getFileName: returns the name of the file including the given method
getRandomMethod: returns a randomly selected method included in the given file at the given revision
isInList: returns whether the given method list (the first parameter) includes the given method (the second parameter). Identifications of methods are based on their signatures and names of their owner files.
add: adds the given method (the second parameter) to the given list (the first parameter)

STEP A-3: Obtaining Syntactic Information of Training Data

After collecting the training data, we obtain the syntactic information of each collected method. We represent a syntactic information of a method as a vector. Each element in this vector indicates the number of appearance of each program element. Hereafter, we call this vector representing the syntactic information of a method **state vector**. The number of appearances of each program element is one element of a state vector. Program elements are, for example, statements (e.g. *if statement*, *switch case*), identifiers, number literals, or symbols (e.g. '.', '==').

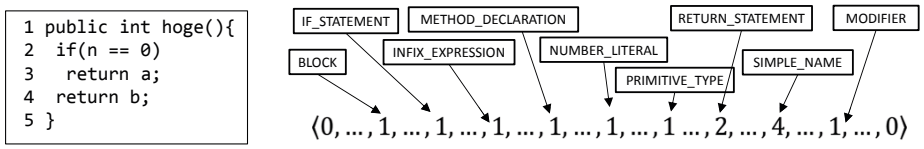
Note that we obtain a state vector in the same way as the existing research [8]. That is, we obtain syntactic information based on AST (Abstract Syntax Tree) generated by JDT[3]. Our proposed technique deals the type of each node of AST as a program element. State vectors that we use in this study are 84-dimensional because JDT defines 84 types of nodes.

Figure 2 illustrates an example of a state vector. Figure 2(a) shows the source code of a method, and Figure 2(b) shows its state vector. In this method, nine kinds of program elements appear in total. Hence, in the state vector of this method, the values of nine elements that appear in the source code are the numbers of their appearances, and the values of the rest 75 elements are 0. For example, the element indicating *return statement* in the state vector is 2 because *return statement* appears twice in the source code. Besides, the element indicating *while statement* is 0 because *while statement* never appears in the method.

The existing techniques identify *Extract Method* refactoring opportunities based on abstract information (e.g. LCOM, control flow and block structure) of source code [9], [10], [11]. However, the accuracy of the prediction by them are not so high, or they detect all the locations that are able to be refactored, which include locations that do not need to be refactored. We believe that using more concrete information of source code syntax should improve the accuracy of identifying refactoring opportunities. Hence, we adopt syntactic information that represents the structure of source code in detail.

Furthermore, the proposed technique performs filtering for *non-refactored methods* based on similarities of state vectors between *refactored methods*. That is, it omits a *non-refactored method* from the training data if the method is similar to any of *refactored methods* in terms of state vector. The rationale behind this is that it would be an obstacle of precise prediction that the training data include *non-refactored methods* which are similar to *refactored methods*. Again, *non-refactored*

[3]http://www.eclipse.org/jdt/



(a) Source Code (b) State Vector

Fig. 2. An Example of a State Vector

method should be *methods that do not require to be refactored*, not just *methods that were not refactored*. It should be more likely that methods might have required being refactored but not refactored by some reasons, if their similar methods were actually refactored. This is the reason that propels us to perform such a filtering.

The proposed technique uses cosine similarity of state vectors for filtering, which is calculated as follows for the given two vectors $\vec{p}$ and $\vec{q}$.

$$\cos(\vec{p},\vec{q}) = \frac{\vec{p}\cdot\vec{q}}{|\vec{p}||\vec{q}|} \quad (2)$$

For each *non-refactored method* gathered in STEP A-2, we calculate cosine similarity between it and all the *refactored methods*. The *non-refactored* method is excluded if there is a *refactored method* whose similarity between the *non-refactored* method exceeds the given threshold. We set 0.95 as the threshold value in this study.

The filtering will leave a hole in the training data. Hence, the proposed technique repeats STEP A-2 and STEP A-3 until the required amount of *non-refactored methods* are collected.

STEP A-4: Constructing a Learning Model

At this point, we have completed obtaining syntactic information of each method in the training data. Finally, the proposed technique learns the syntactic information of each method supervised by the information whether *Extract Method* has been applied to it, which results in constructing a learning model. The proposed technique predicts whether *Extract Method* should be applied to a given method or not when the constructed learning model is given the syntactic information of the given method. CurrentlyIn our implementation, we use a data mining tool, Weka[4], to construct a learning model with its default setting.

IV. Experiment

We implemented the proposed technique as a tool, and conducted an experiment on five projects shown in Table I. For each target project, we obtained all the *refactored methods* with the procedures described in the previous section. We also obtained the same number of *non-refactored methods* as *refactored-methods*. We used those methods as a dataset in this experiment. As described in the previous section, the procedure to construct a dataset is fully automated.

In this experiment, we investigated the following two research questions in order to evaluate our proposed technique.

[4]http://www.cs.waikato.ac.nz/ml/weka/

RQ1: is it possible to identify methods to which *Extract Method* should be applied with machine learning techniques?

RQ2: does the length of development histories affect the accuracy of the proposed technique?

We describe our answers for these research questions in the remainder of this section.

RQ1: is it possible to identify methods to which Extract Method should be applied with machine learning techniques?

This experiment adopts the cross-validation. The cross validation divides the dataset into k blocks, and evaluates the accuracy of prediction by using these blocks. Note that the number of methods in each block is almost the same. In the cross-validation, we use $k-1$ blocks as the training data, and the remaining one block as the test data. More concretely, we construct a learning model by learning $k-1$ blocks. Then, we adopt the learning model to the remaining one block and measure the accuracy. This process is repeated k times, with each of the k blocks used exactly once as the test data. The average of k results is calculated to represent the complete accuracy of predictions. We set $k = 10$ in this experiment.

We used two indicators below as measures for evaluation.

Precision: the proportion of *refactored methods* out of methods predicted as *refactored methods* by the learning model

Recall: the proportion of methods that are correctly predicted as *refactored methods* by the learning model in all the *refactored methods*

Not only *Precision* but also *Recall* is measured in this experiment because *Recall* is also important for techniques finding refactoring opportunities. If a technique finds refactoring opportunities with high *Precision* and low *Recall*, most of the suggested refactorings by it are reasonable refactoring targets whereas it misses many refactoring opportunities. Finding refactoring opportunities with high *Precision* and high *Recall* is required to be practical.

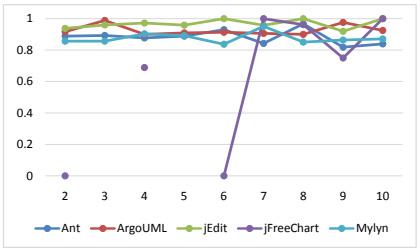
We use five algorithms (J48, Random Forest, Bayesian Network (BayesNet), Support Vector Machine (SVM), Logistic Regression (Logistic)) to construct learning models, and evaluate each learning model with the cross-validation.

TABLE I
Target Projects

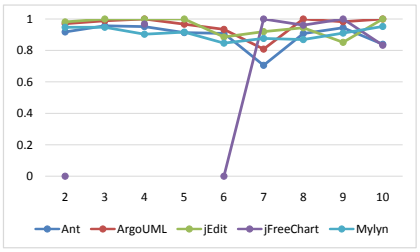
Project	# of Revisions	LOC (Latest)	# of Dataset	
			Refactored	Non-refactored
Ant	12,783	260,624	766	766
ArgoUML	17,788	370,750	735	735
jEdit	7,473	187,166	533	533
jFreeChart	916	327,865	90	90
Mylyn	8,414	166,149	490	490

TABLE II
THE PREDICTION RESULTS

algorithm	Ant		ArgoUML		jEdit		jFreeChart		Mylyn	
	Precision	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*
J48	0.90	0.91	0.95	0.96	0.96	0.95	0.93	0.97	0.91	0.89
RandomForest	0.89	0.95	0.94	0.98	0.96	0.98	0.89	0.98	0.89	0.94
SVM	0.95	0.71	0.95	0.81	0.99	0.88	0.91	0.57	0.95	0.76
BayesNet	0.90	0.79	0.96	0.89	0.97	0.97	0.86	0.97	0.93	0.86
Logistic	0.91	0.86	0.96	0.95	0.97	0.93	0.91	0.95	0.93	0.90



(a) *Precision*



(b) *Recall*

Fig. 3. Results on Different Length of Learned Periods (with RandomForest)

Table II shows the prediction results with the five algorithms. As shown in this table, the proposed technique was able to identify *refactored methods* with accuracy, 86–97% *Precision* and 71–98% *Recall*. SVM has high *Precision* values for all the projects whereas it reports lower *Recall* values than other algorithms. The table also tells that the algorithms based on decision tree (J48 and RandomForest) record over 89% for both of *Precision* and *Recall*. The results indicate that the proposed technique has an affinity with algorithms based on decision tree.

Besides, very high *Recall* means that the proposed technique is able to identify almost all the *refactored methods*. Hence, our tool is useful to identify the candidates of *Extract Method* refactoring opportunities in the source code. Then, users can select efficiently the methods that they want to refactor among the candidates.

Consequently, our answer to RQ1 is that it is possible to identify refactoring opportunities by using machine learning techniques. Especially, the proposed technique works well with algorithms based on decision tree.

RQ2: does the length of development histories affect the accuracy of the proposed technique?

Our answer to RQ1 revealed that we can predict where we should refactor through learning refactorings in the past. However, the proposed technique will be affected by the length of the development history of the target projects. Intuitively, it seems likely that the accuracy of prediction gets better as the development history gets longer. This is because there exist many refactoring to be learned. On the other hand, the proposed technique can be adopted in young projects if it can perform good prediction from a short history.

We aim to investigate the relationship between the length of the development histories and the accuracy of the proposed technique in this experiment. Here, we divide the development period of each target project into 10 equivalent sub-periods. We then apply the proposed technique for each of the second or later sub-periods. Suppose we are targeting a sub-period t, then we build a prediction model by learning all the sub-periods before t. t is then used to evaluate the model built from its anterior sub-periods.

Figure 3 illustrates the results of this experiment. The X-axes show the targeted sub-periods in chronological order, which means that the sub-period i is earlier than $i+1$. This figure shows only the results with RandomForest because similar results were provided with other algorithms. It also should be noted that the results for jFreeChart is not continuous. This is because there is no *refactored methods* in some of the sub-periods of jFreeChart. As we can see from this figure, the accuracy of prediction seems not to be affected how long periods are used for learning. These results indicate that the proposed technique does not require long development history. In other words, the proposed technique can be used not only in mature projects but also young ones.

The answer to RQ2 is that learning longer histories produces do not always better results, which means that the proposed technique works even in young projects if their development histories include *refactored methods*.

V. DISCUSSION

A. Characteristic of Refactoring Opportunity

We investigated the characteristics of refactoring opportunities for each target project. As a result, we obtained two knowledge.

First, long methods are likely to be refactored for all the target projects. We infer developers shorten long methods by *Extract Method* in order to improve the understandability.

Second, program elements that are considered important to identify refactoring opportunities are not necessarily common to different projects. We investigated what kinds of program elements are regarded as important in learning models constructed with RandomForest by examining the structure of the models. TABLE III shows a ranking of the program elements that are regarded as important for each project. As shown

TABLE III
RANKING FREQUENCIES OF PROGRAM ELEMENTS (WITH RANDOMFOREST)

Project	Top-1	Top-2	Top-3
Ant	METHOD INVOCATION	VARIABLE DECLARATION STATEMENT	CATCH CLAUSE
ArgoUML	METHOD INVOCATION	NUMBER LITERAL	RETURN STATEMENT
jEdit	SIMPLE NAME	METHOD INVOCATION	IF STATEMENT
jFreeChart	METHOD INVOCATION	THIS EXPRESSION	SIMPLE TYPE
Mylyn	METHOD INVOCATION	SWITCH CASE	SYNCHRONIZED STATEMENT

from this table, we revealed that METHOD INVOCATION elements are important to identify refactoring opportunities in learning models for all the target projects. More concretely, methods that include many METHOD INVOCATION elements are likely to be refactored. On the other hand, there are some cases that program elements considered much important for one project are not regarded as important for other projects. For example, the number of IF STATEMENT elements has a significant impact on identifying refactoring opportunities for jEdit while it does not affect the identifications for ArgoUML.

Consequently, it is reasonable to identify refactoring opportunities by projects by using machine learning techniques because refactoring opportunities differ from project to project.

B. Finding Refactoring Opportunities with Machine Learning Techniques

The authors mention two advantages of finding refactoring opportunities by using machine learning techniques. First, it is possible to find refactoring opportunities by characteristic of projects. Techniques based on pre-set criteria or pre-set conditions are difficult to identify refactoring opportunities that depend on the projects because refactoring opportunities differ from project to project. However, it is possible to identify refactoring opportunities that are useful for projects without pre-set criteria by learning refactorings that were conducted in the past.

Second, it prevents developers from overlooking refactoring opportunities. Developers might overlook refactoring opportunities in manual identification process because the amount of source code is usually huge. For example, developers frequently extract duplicate code (code fragments similar to each other) in different methods and integrate it as a method during software development. However, it may happen that they overlook a part of the instance of code duplications if they identify it manually. There is a research report that developers occasionally overlook a part of duplicate code to be refactored when they integrate it by *Extract Method* [12]. However, by learning refactorings conducted in the past with machine learning techniques, the locations that developers overlooked can be detected. Consequently, finding refactoring opportunities by using machine learning techniques are useful because they can detect locations similar to the locations that were refactored in the past.

VI. CONCLUSIONS AND FUTURE WORK

We proposed a technique to automatically find *Extract Method* refactoring opportunities by using machine learning techniques. The proposed technique finds refactoring opportunities flexibly in accordance with projects' and developers' features because it is possible to identify refactoring opportunities based on information of refactorings that were conducted actually in the past. The proposed technique learns refactorings that were conducted in the past by analyzing development histories of target software, and it constructs models to identify locations to be refactored in source code.

To evaluate the proposed technique, we conducted an experiment on five open source projects. As a result, we confirmed that the proposed technique was able to identify *Extract Method* refactoring opportunities. Especially, the technique worked well with the algorithms based on decision tree.

Our future works are as follows:

- conducting experiment on various projects,
- changing parameters of Weka,
- using different data mining tools,
- using all the *Extract Method* refactorings that were conducted in the past, which include refactorings that are not able to be detected by Kenja, and
- investigating whether the proposed technique can find refactoring opportunities by characteristics of developers.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI JP25220003

REFERENCES

[1] E. Van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," in *Proceedings of WCRE*, 2002.

[2] K. Hotta, Y. Higo, and S. Kusumoto, "Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph," in *Proceedings of CSMR*, 2012.

[3] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the Relation of Refactorings and Software Defect Prediction," in *Proceedings of MSR*, 2008.

[4] M. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," in *Proceedings of METRICS*, 2005.

[5] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code Smell Detection: Towards a Machine Learning-Based Approach," in *Proceedings of ICSM*, 2013.

[6] E. Murphy-Hill, C. Parnin, and A. Black, "How We Refactor, and How We Know It," *IEEE TSE*, vol. 38, no. 1, 2012.

[7] A. Broder, "On the resemblance and containment of documents," in *Proceedings of SEQUENCES*, 1997.

[8] H. Murakami, K. Hotta, Y. Higo, and S. Kusumoto, "Predicting Next Changes at the Fine-Grained Level," in *Proceedings of APSEC*, 2014.

[9] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of CSMR*, 2001.

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities," in *Proceedings of CSMR*, 2009.

[11] D. Silva, R. Terra, and M. T. Valente, "Recommending Automated Extract Method Refactorings," in *Proceedings of ICPC*, 2014.

[12] Y. Higo, K. Hotta, and S. Kusumoto, "Enhancement of CRD-based Clone Tracking," in *Proceedings of EVOL/IWPSE*, 2013.