Copyright ⓒ 2018 KSII

Automatic identification of Java Method Naming Patterns Using Cascade K-Medoids

Tae-young Kim[1], Suntae Kim[1], Jeong-Ah Kim[2*], Jae-Young Choi[3], Jee-Huong Lee[3], Youngwha Cho[3], Young-Kwang Nam[4]

[1]Dept. of Software Engineering, CAIIT, Chonbuk National University
567 Baekje-daero, deokjin-gu, Jeonju-si, Jeollabuk-do 54896, Republic of Korea
[2*]Department of Computer Education, Catholic Kwandong University, Beomil-ro 579 beon-gil, Kangneung-Si, Kangwon-Do, Republic of Korea
[3]College of Information and Communication Engineering, SungKyunKwan University, 2066 Seobu-Ro, Jangan-Gu, Suwon, Gyeonggi-Do, Republic of Korea
[4] Department of Computer and Telecommunications, Yonsei University
Gangwon-do, Wonju-si, Yeonsedae-gil, 1
*Corresponding Author: Jeong-Ah Kim
[e-mail: {rlaxodud1200, stkim }@jbnu.ac.kr, clara@ckd.ac.kr, {jaeychoi, john, choyh2285}@skku.edu, yknam@yonsei.ac.kr]

Received September 29, 2017; revised January 19, 2018; accepted February 13, 2018; published February 28, 2018

Abstract

This paper suggests an automatic approach to extracting Java method implementation patterns associated with method identifiers using Cascade K-Medoids. Java method implementation patterns indicate recurring implementations for achieving the purpose described in the method identifier with the given parameters and return type. If the implementation is different from the purpose, readers of the code tend to take more time to comprehend the method, which eventually affects to the increment of software maintenance cost. In order to automatically identify implementation patterns and its representative sample code, we first propose three groups of feature vectors for characterizing the Java method signature, method body and their relation. Then, we apply Cascade K-Medoids by enhancing the K-Medoids algorithm with the Calinski and Harrabasez algorithm. As the evaluation of our approach, we identified 16,768 implementation patterns of 7,169 method identifiers from 50 open source projects. The implementation patterns have been validated by the 30 industrial practitioners with from 1 to 6 years industrial experience, resulting in 86% of the precision.

Keywords: Implementation Pattern, Java Method Signature, Cascade K-Medoids

A preliminary version of this paper was presented at APIC-IST 2017, and was selected as an outstanding paper.

http://doi.org/10.3837/tiis.2018.02.020 **ISSN : 1976-7277**

1. Introduction

In building SW systems with the Java language, a method is a minimal unit of implementation of system's functionality. A method is generally composed of two parts: a method signature part of declaring an identifier, parameters and a return type, and a method body part with realizing the purpose of the method. When a code reader is faced with the method signature, they expect an implementation associated with the method signature. The expected implementation is defined as implementation patterns indicating recurring implementations for achieving the purpose described in the method identifier with the given parameters and return type[1]. In case that the method is furthering than the expected implementation pattern, the readers should take a longer time to comprehend the method, which eventually causes the increment of software maintenance cost[2].

There have been several research on addressing the above issue. Those can be classified into two research streams. First, some of the research suggested implementation patterns of writing a code snippet of recurring programming situations. While this research tries to suggest best practices for coding, the approaches heavily rely on their prior experience without any statistics. Another research stream proposed an automatic approach to validating method identifiers based on datamining algorithms such as n-gram or association rule mining. However, they only focus method identifiers without examining the method body part that realizes an intention specified in the method identifier.

In order to address the issue, we suggest an automatic approach to extracting Java method implementation patterns associated with method identifiers using the Cascade K-Medoids algorithm. We first propose *Behavior*, *Signature* and *Relation Vectors* for characterizing the Java method signature, method body and their relation. Then, we apply the Cascade K-Medoids algorithm by enhancing K-Medoids with Calinski and Harrabasez in order to extract appropriate number of implementation patterns of a specific identifier, and its representative sample code of each pattern. For validating our approach, we identified 16,768 implementation patterns of 7,169 method identifiers from 50 open source projects. The implementation patterns have been validated by the 30 industrial practitioners with from 1 to 6 years industrial experience, resulting in 86% of the precision. Also, we intensively discuss the gap between statistics of our approach and human validation.

The rest of the paper is structured as follows: Section 2 presents related work on implementation patterns and automatic approach to validate Java method identifiers. Section 3 presents our approach from identifying implementation patterns. Section 4 presents an evaluation of our approach and discuss the result. Section 5 concludes the paper with future work.

2. Related Work

This section presents prior work that targeted the identification of implementation patterns of source code and validation of the method identifiers. Kent Beck first defined the implementation pattern as a "catalog of the common problems of programming and the features of Java that addresses those problems"[1]. He introduced best practices to write a source code in various situations such as *Control Flow, Choosing Message, Exception* and so forth. The implementation patterns cover a code snippets as well as several methods that handle those situations. Gil and Maman[3] formally defined 27 implementation patterns such

as *Designator, Pool, Data Manager* and so forth, investigated them from the 15 popular open source libraries, and presented how the patterns exist on the libraries.

The above mentioned approaches propose the implementation patterns based on the authors' experience without validating them, which is considered as best practices associated with a specific situation. In this paper, we focus on the implementation patterns associated with the method identifier. The method identifier is similar to the situation in the code, because the situation indicates what is happening in a particular place, and the method identifier denotes the purpose that the method body should realize. However, implementation patterns of method identifiers and situation is different. While the former refers to recurring patterns to realize the method identifiers, the latter refers to recurring patterns to cope with the specific situation. This paper focuses on the former.

Some of the other research tried to automatically validate the method identifiers. Kashiwabara *et al*. suggested an approach to recommending appropriate verb part in the method identifier by using an association mining algorithm[4] They collect various identifiers (e.g., class identifier, field identifier, argument types, sub method identifiers etc.) in the source code first, and extract relations between the verb part of the method and the collected identifiers by applying the association rule mining algorithm. Then, they recommend an appropriate verb part of a method for the developers. Takayuki *et al*.[5] suggested the n-gram model[6] based approach to suggest appropriate method identifiers. While they suggested not only the verb part, but also full method identifiers, the precision of the suggested method identifiers might not be appropriate. This is because they only examine the method identifier part, without the body part of the method.

In order to verify the method identifier, Deißenbock[7] constructs the identifier data dictionary, finds the homonym and synonym of the identifier word of the corresponding method, and verifies the conciseness and consistency of the identifier based on the homonyms and the synonyms. However, in order to verify the consistency of identifiers by searching homonyms and synonyms, the experts should connect these concepts manually. In a similar study, Lawrie [8] also analyzed homonyms and synonyms, and constructed patterns using natural language processing to identify consistency. In addition, they used WordNet[9] to automatically identify synonyms to improve the research shortcomings of Deißenbock *et al.*[7]. However, there is a limitation that the range of synonyms is too broad and the accuracy of their approach is low, because it does not analyze the structure of the sentence. The above-mentioned studies solve the problem by taking only the method signature as a main feature without deeply considering the relation between the implementation part of the method and the identifier.

3. IDENTIFYING IMPLEMENTATION PATTERNS USING CASCADE K-MEDOIDS

This section presents an automatic approach to identifying implementation patterns associated with Java method identifiers from open source projects. The approach consists of three steps as shown in **Fig. 1**. The first step is preprocessing of the method identifiers with AST(Abstract Syntax Tree) parsing, exceptional method filtering and POS(Part of Speech) tagging. In the second step, behavioral, signature and relation feature vectors that we suggest are automatically extracted from the code, and they are compiled according to the POS of each method. At the last step, implementation patterns and a representative sample code of each

pattern are identified by using the Cascade K-Medoids algorithm. We describe each step in more detail from the following subsections.

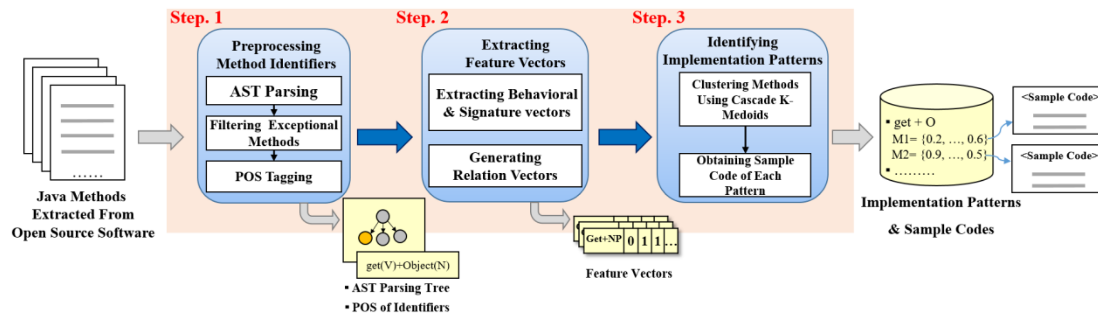


Fig. 1. A Process for Identifying Implementation Patterns and Sample Code

3.1 Step 1: Preprocessing Method Identifiers

The first step is to carry out preprocessing of the method identifiers. More detailed steps for this preprocessing are presented in **Fig. 2**. It starts with parsing AST of the Java method in order to separately access method signature and method body parts. The AST parsing tree as shown in the figure is shaped as a tree structure composing of each token and operators, and it enables one to access all elements of the method. To build the AST parsing tree, we applied the Eclipse AST parser[10].

As the second step, the method identifiers which do not follow the Java naming convention[11] are filtered out. According to the Java naming convention, Methods should be verbs(a verb phrase) and should start with a lower case. As an example, the getName() method starts with the verb *get* followed by the objective Name, so it makes a verb phrase. However, some of the method identifiers do not follow the convention, but it is broadly accepted. For example, main(), length() and size() are widely used method identifiers, but they do not observe the convention. These methods are classified as an Idiom. In our previous research, we statistically identified the idioms[12]. Based on the idiom list, the idiom methods are excluded in identifying implementation patterns.

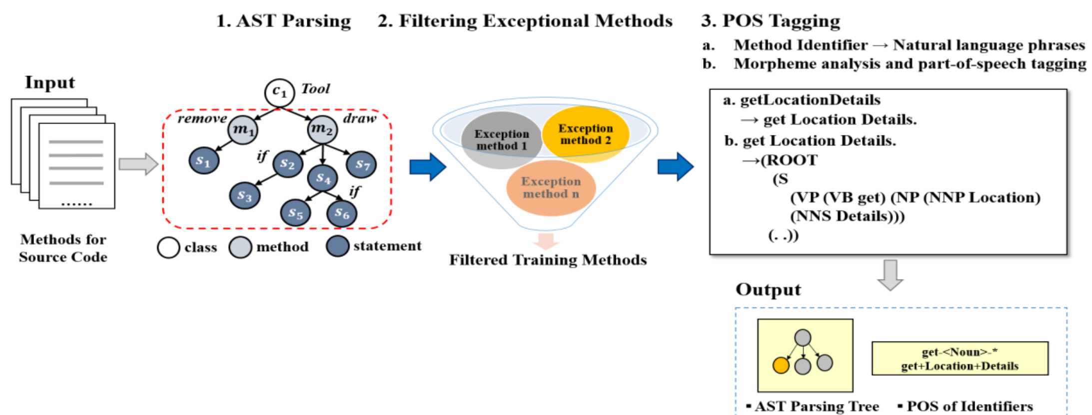


Fig. 2. Steps for Preprocessing of Method Identifiers

The last step of the preprocessing is to carry out POS tagging for the method identifiers using the NLP Parser[1]. In order to conduct the POS tagging, the method identifier is tokenized, put a blank between the tokens, and append a period at the end of the tokens to make a complete sentence. Then, the NLP parser† syntactically analyzes the sentence and produces the parsing tree using Penn Tree Bank. Based on the result, we can obtain the POS of each word composing of the method identifier. For example, the getLocationDetails() method (see the right box in **Fig. 2**) is converted into 'got Location Details.', and the Stanford parser produces the parsing tree as shown in the box. Finally the parser produces the verb part get and the objective part Location Details. The result is converted to the "get + O(Objective)" to compile the feature vectors of similar style of method identifiers such as getName(), getPerson() and so forth.

3.2 Step 2: Extracting Feature Vectors

The purpose of the second step is to define feature vectors for characterizing the method signature and body to extract implementation patterns. The feature vector is classified into three categories: Signature, Behavioral and Relation vectors. We proposed these feature vectors in our previous work[14]. In this paper, we summarize the feature vectors in short.

While the signature vectors capture characteristics of the method signature part including parameters and return type, the behavioral vectors characterize key features of the method body implementation. Also, the relation vectors capture the relation between the method signature and method body part, focusing on where the words in the method identifiers or parameters are used in the method body. **Fig. 3** illustrates the relation among three feature vectors.

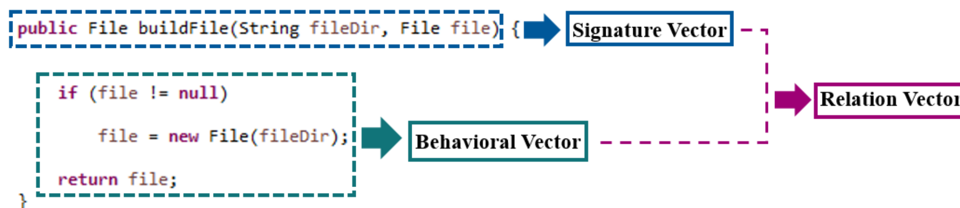


Fig. 3. Relation of Three Feature Vectors

Table 1. Signature Vectors

Features	Type
ReturnType(void, Boolean, int, String, Object)	Factor
Parameters	Numeric

Table 2. Behavioral Vectors

Features	Description	Type
existFieldRead	read a class field	Boolean
existFieldWrite	write data to a class field	Boolean
existTypeManipulator	There is a type casting operator	Boolean
existLocalReader	read a local variables defined in the method or passed as parameters	Boolean

[1] In this paper, we have adopted Stanford Parser[13] because it is highly accurate for parsing natural language sentences and broadly used in NLP. In addition, it is publicly available, well-documented and stable.

existLocalWriter	write data to a local variables	Boolean
existArrayReader	read array type local variables	Boolean
existArrayWriter	write array type data to local variables	Boolean
createInstance	create diverse types of instances such as String, Array, Custom and Primitive	Factor
Lines	the number of the method body	Numeric

Signature vectors are for representing characteristics of a method signature, composing of *ReturnType* and *Parameters* as shown in **Table 1**. ReturnType indicates a type of return such as *void*, *Object* and *String*. Thus, the data type of this feature is the Factor type. On the other hand, *Parameter* denotes the number of parameters, represented in the *Numeric* type.

Behavioral vectors are intended to capture characteristics of a method body as shown in **Table 2**. Most of the feature vectors are the *Boolean* type, while the type of the *createInstance* feature is the *Factor* type to capture different types of object or value creations. Also, the Lines feature as a numeric type denotes the number of a method body line.

Relation vectors indicate a set of features that express existence of the objective part of a method identifier and parameters in the method body. Thus, a data type of all feature vectors is *Boolean*. We grouped them into two sub-groups. First, the *Include* relation vector group checks if the objective part of the method identifier or parameters are existed in the diverse elements in the method body as shown in **Table 3**. The *Target* relation vector group checks if the objective part and parameters are used in the elements of the method body part as shown in **Table 4**.

Fig. 4 illustrates an example of the relation vector *Obj_InIfBranch_Condition* and *Obj_InCreateObjects*. The *Obj_InIfBranch_Condition* feature vector checks if the objective part of the method identifier exist in the condition part of the *if* statement in the method body part. In the example, the feature is *True* in this case because the File is located in the condition part of the *if* statement. In addition, the *Obj_InCreateObjects* feature checks if the the objective part (i.e., File) exists in the create instance statement. Thus, it becomes *True* in this example.

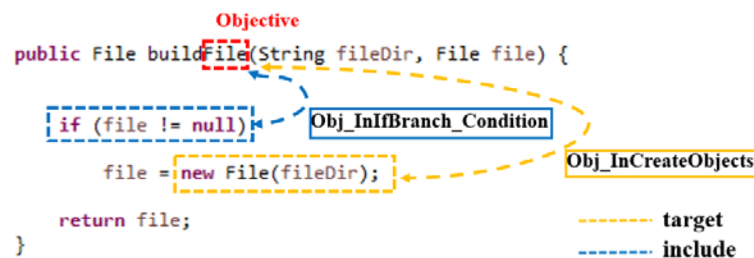


Fig. 4. Examples of the Relation Vectors

Table 3. *Include* Relation Vectors

Relation	Feature	Description
Objective Method Body	Obj_InReturnVariable	The obj. exists in the return variable
	Obj_InReturnType	The obj. existes in the return data type
	Obj_InParameter	The obj. exists in the parameter name.
	Obj InParameterType	The obj. exists in the parameter type.

	Obj_InSubMethodName	The obj. exists in the sub method name .
	Obj_InSubMethodParam	The obj. exists in the sub method parameters
	Obj_InIfBranch_Condition	The obj. exists in the condition part of the if statement.
	Obj_InIfBranch_Body	The obj. exists in the body part of the if statement
	Obj_InLoop_Body	The obj. exists in the body part of the loop (i.e., for or while) statement.
Parameters Method Body	Param_InReturnVariable	The param. exists in the return variable.
	Param_InReturnType	The param. exists in the return type.
	Param_InSubMethodName	The param. exists in the sub method name.
	Param_InSubMethodParam	The param. exists in the sub method parameters.
	Param_InIfBranch_Condition	The param. exists in the condition part of the if statement.
	Param_InIfBranch_Body	The param. exists in the body part of the if statement.
	Param_InLoop_Condition	The param. exists in the condition part of the loop statement.
	Param_InLoop_Body	The param. exists in the body part of the loop statement.

Table 4. *Target* Relation Vectors

Relation	Feature	Description
Objective Method Body	Obj_InCreateObjects	The obj. exists in the create instance statement.
	Obj_InFieldReader	The obj. exists in the reading a class field code.
	Obj_InFieldVariable	The obj. exists in the class field variable.
	Obj_InTypeManipulator	The obj. exists in the class casting statement.
	Obj_InLocalVariable	The obj. exists in the local variable
	Obj_InArrayVariable	The obj. exists in the local array variable
Parameters Method Body	Param_InCreateObjects	The param. exists in the create instance statement.
	Param_InFieldReader	The param. exists in the reading a class field code.
	Param_InFieldVariable	The param. exists in the field variable.

	Param_InTypeManipulator	The param. exists in the class casting statement.
	Param_InLocalReader	The param. exists in the reading local variable.
	Param_InLocalVariable	The param. exists in the local variable.
	Param_InArrayVariable	The param. exists in the local array variable.

3.3 Step 3: Identifying Implementation Patterns

This step aims at identifying implementation patterns from the feature vectors grouped by the similar styles of method identifiers by using the Cascade K-Medoids algorithm. The Cascade K-Medoids algorithm that we suggest resolves two issues: pointing at the specific method as a representative method of the implementation pattern and finding appropriate number of clusters(i.e., the number of implementation patterns).

In order to handle the first issue, we selected the K-Medoids algorithm[15] as a base algorithm. To search a representative sample code in a cluster, the centroid based clustering algorithms such as K-Means[16] and K-Medoids algorithms is appropriate. This is because they compute the centroid and medoid as a center of a cluster, which can be considered as a representative method of an implementation pattern. The key difference between the K-Means and K-Medoids algorithm is that a centroid of K-Means is a virtual point just computed by an arithmetic average of feature vectors, while a medoid of K-Medoids indicates a real entity positioned at the center of the cluster. Thus, the K-Medoids algorithm is selected as an appropriate base algorithm to find a representative sample method of the implementation pattern.

The main lack of the K-Medoids algorithm is that we need to specify the K value (i.e., the number of clusters) in advance, which is the second issue. To tackle this issue, we applied the *Calinski and Harabasz* algorithm [17] to find the optimal K by maximizing inter-cluster entity variance and minimizing intra-cluster entity variance based on **Equation 1**. In the equation, *K, B(K)*, *W(K)* and *n* indicate the number of clusters, the inter-cluster variance, the intra-cluster variance and the number of all entities respectively. The Calinski and Harabasz algorithm initially starts with *K = 2* and computes *CH(2)*, then gradually increases the K to find the optimal number of clusters.

$$\mathrm{CH}(K) = \frac{B(K)/(K-1)}{} \qquad (1)$$

In addition, we applied the following distance function to compute a distance between two entity x_i and x_j as shown in **Equation 2**. As the feature vectors are composed of the compound data types including *factor*, *boolean* and *numeric*, the similarity function should be handling it according to the data types as shown in **Equation 3**. Also, the numeric data types is normalized because they have a different range among numeric types of feature vectors[19].

$$\mathrm{d}(x_i, x_j) = \frac{1}{\mathrm{p}} \sum_{k=1}^{p} sim(x_{ki}, x_{kj}) \qquad (2)$$

$$sim(x_{ki}, x_{kj}) = \begin{cases} 1 \; if \; type = factor \; or \; boolean, and \; x_{ki} = x_{kj} \\ 0 \; if \; type = factor \; or \; boolean, and \; x_{ki} \neq x_{kj} \\ \dfrac{|x_{ki} - x_{kj}|}{\max x_k - \min x_k} \; if \; type = numeric \end{cases} \quad (3)$$

4. EVALUATION

This section describes the implementation patterns and a representative sample method for each implementation pattern by using the suggested approach, then presents a result of a user study for checking validity for the implementation patterns.

4.1 Implementation Patterns and a Representative Method of Each Pattern

In order to identify the implementation patterns and a sample method of each pattern suggested in Section 3, we first collected 50 open source projects listed in **Table 5**. The open source projects covers build tools, XML frameworks, web/servlet containers and diverse GUI-based applications. The full list of the open source projects are presented in Appendix A. The open source projects contain the 43,297 classes with 452,816 methods.

Based on the feature vectors extracted from the methods of the open source projects, we have identified 16,768 implementation patterns for 7,169 method identifier styles that start with the same verb followed by the same POS. **Table 6** shows a selected set of implementation patterns and the number of methods. The complete list of the patterns is presented in **Table 11** and **Table 12** in Appendix B. According to **Table 6**, the *get_NP()* style method identifiers such as *getName()* and *getLocation()* exist 101,721 methods in the data set, and four implementation patterns are identified from the methods. The last column of the table shows the number of methods included in each implementation pattern.

Table 5. Open Source Project List for Training

Project	Version	Description	# of Method
Ant	1.9.2	Build Tool	4,694
Activemq	5.4.1	JMS Messaging Engine	4,651
ArgoUML	0.32.0	UML Modeling Tool	2,361
Apache-commons math	3.4.1	Math Library	4,266
Apache tomcat	8.5.16	Servlet Engine	5,811
Apache Lucene	6.0.1	Search Engine	6,487
Apache Jena	2.12.1	RDF/SPARQL Framework	2,331
Castor	1.1.2	XML framework	13,900
Cglib	2.2.2	Code Generation Library	1,709
Dom4j	1.5.1	XML Framework	2,521
Jboss	6.1.0	Web Application Server	3,348
Jgroup	2.6.21	Group communication toolkit	5,609
Jedit	5.3.0	Text Editor	18,124
Spring-Framework	2.5.6	Framework for Java based Enterprise Applications	53,591
jackrabbit	2.14.1	Content Repository	5,124
JHotDraw	5.2.0	GUI Toolkit	4,685

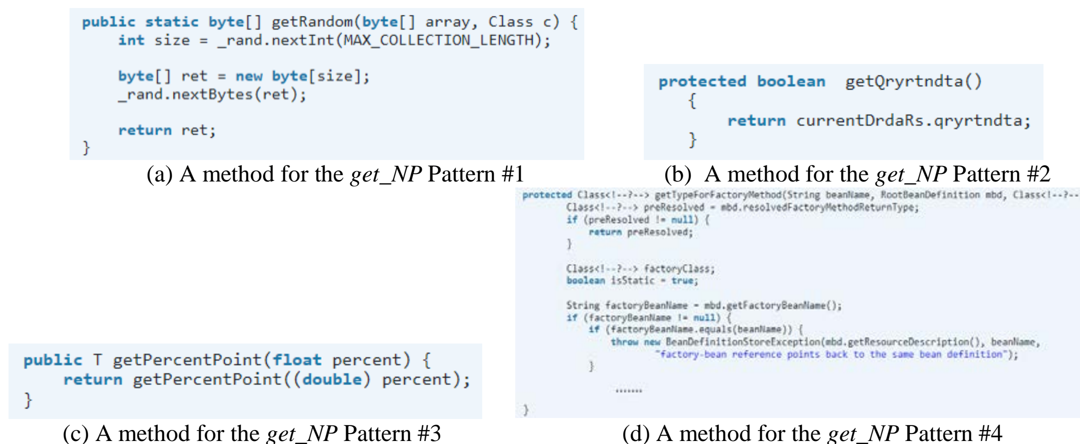
Table 6. List of Implementation Patterns and Num. of Methods

Method Naming	# of Method	Im. Patterns	# of Methods of Each Pattern
get_NP	101,721	#1	37,586
		#2	36,486
		#3	24,498
		#4	3,151
set_NP	45,762	#1	17,964
		#2	14,312
		#3	13,486
add_NP	5,853	#1	3,127
		#2	1,649
		#3	1,077
equals_NP	781	#1	409
		#2	372
create	665	#1	339
		#2	316
reset	324	#1	164
		#2	160

Fig. 5 and **Fig. 6** present the representative methods of the *get_NP()* and *add_NP()* style method identifier's implementation patterns. The most of representative methods of each implementation pattern are conspicuously different each other, and quite frequently discovered in the source code. The methods of the Figure. **Fig. 5(d)** and **Fig. 6(c)** are abbreviated due to the space limit.

4.2 Checking Validity of the Implementation Patterns

This section describes a user-study experiment for validating the identified implementation patterns. This step is crucial to validate if the identified patterns are reasonable for human practitioners, because the previous subsection only shows the implementation patterns and its method included statistically. In order to carry out this experiment, we collected 1 ~ 6 years industrial practitioners as shown in **Table 7**.



(a) A method for the *get_NP* Pattern #1
(b) A method for the *get_NP* Pattern #2
(c) A method for the *get_NP* Pattern #3
(d) A method for the *get_NP* Pattern #4

Fig. 5. The Representative Method of Each Implementation Pattern of the *get_NP* style

```
public boolean addAll(int index,Collection<!--? extends Instr--> c){
ListIterator<instr> iterator=listIterator(index);
if (c.isEmpty())   return false;
for (  Instr instr : c) {
  iterator.add(instr);
}
return true;
}
```

(a) A method for the *add_NP* Pattern #1

```
private void addToRecent(String s) {

        for (int i = 0; i < iMaxRecent; i++) {
            if (s.equals(sRecent[i])) {
                return;
            }
        }

        if (sRecent[iRecent] != null) {
            mRecent.remove(iRecent);
        }

        sRecent[iRecent] = s;

        if (s.length() > 43) {
            s = s.substring(0, 40) + "...";
        }

        JMenuItem item = new JMenuItem(s);
            .......
 }
```

(b) A method for the *add_NP* Pattern #2

```
private final void addLast(Object node) {
        objects.addLast(node);

 }
```

(c) A method for the *add_NP* Pattern #3

Fig. 6. The Representative Method of Each Implementation Pattern of the *add_NP* style

Table 7. Subjects for Our Experiment

Work Year	Num. of Subjects
5~6 years	8
3~4 years	18
1~2 years	14

We made an on-line questionnaire to check the validity of each pattern. We selected 30 sample method identifier styles and showed each method identifier's patterns with the representative method. Each subject responded the validity of each pattern based on the 4 point scale with the criteria described in **Table 8**.

Based on the subjects, questionnaires and evaluation criteria, we established two research questions as follows:

- RQ1: How appropriate are the identified implementation patterns?
- RQ2: How different are statistics and human-validation of implementation patterns?

The followings describe the experiment setting and the analysis of the results.

Table 8. Validation Criteria of Each Pattern

Point	Designation	Description
4	Very appropriate	The implementation pattern clearly and uniquely represent the method styles associated with method identifier, parameters and return type.
3	appropriate	The implementation pattern appropriately represent the method styles including the verb part.
2	normal	The implementation pattern only captures the characteristics of the verb part of the method identifier
1	Inappropriate	The implementation pattern has some significant weakness to represent the implementation pattern.

Table 9. List of Implementation Patterns and Questionnaire Average Score

Method Identifier	Im. Patterns	Mean Score
get_NP	#1	2.69
	#2	3.13
	#3	2.98
	#4	3.03
set_NP	#1	3.27
	#2	3.23
	#3	3.14
add_NP	#1	2.83
	#2	1.03
	#3	2.21
equals_NP	#1	2.91
	#2	2.44
create	#1	3.41
	#2	3.32
reset	#1	3.76
	#2	3.01

4.2.1. RQ1: *How appropriate are the identified implementation patterns?*

This research question is intended to check if the identified implementation patterns are appropriate. All subjects responded the extent of the appropriateness of the patterns based on the evaluation criteria. We evaluate the patterns with a mean score of less than 2 point as inappropriate and the patterns with a mean score of more than 2 point as appropriate. All implementation patterns' score are presented in Appendix B, and **Table 9** shows the partial set of results of the experiment.

For most of these implementation patterns for each method identifier style, the subjects gave over 2.0 points on average to each implementation pattern. Particularly, the *set_NP()* and *reset()* identifiers obtained 3.23 and 3.4 on average respectively, which is considered as very appropriate implementation pattern. However, the second implementation pattern of the *add_NP()* style method identifiers only obtained 1.0 point. This is because the representative method addToRecent(String s) is somehow GUI specific code with the JMenuItem class as shown **Fig. 6(b)**. Except the implementation patterns, most of the implementation patterns were considered as appropriate patterns.

In addition, we computed the precision to figure out appropriateness of the implementation patterns more. The *precision* is computed by **Equation 4**[18]. In the experiment, we showed 66 implementation patterns to the subjects, and 57 patterns obtained more than 2.0 point. Thus, we obtained 0.86 precision, which is considered as 86% of implementation patterns is considered as appropriate patterns. It should be noted that it is impossible to compute the *recall* of the implementation patterns, because the subjects should manually inspect all source code to find patterns and should discuss the appropriateness of each pattern. Thus, it is too time-consuming and even the appropriateness of the patterns is not guaranteed.

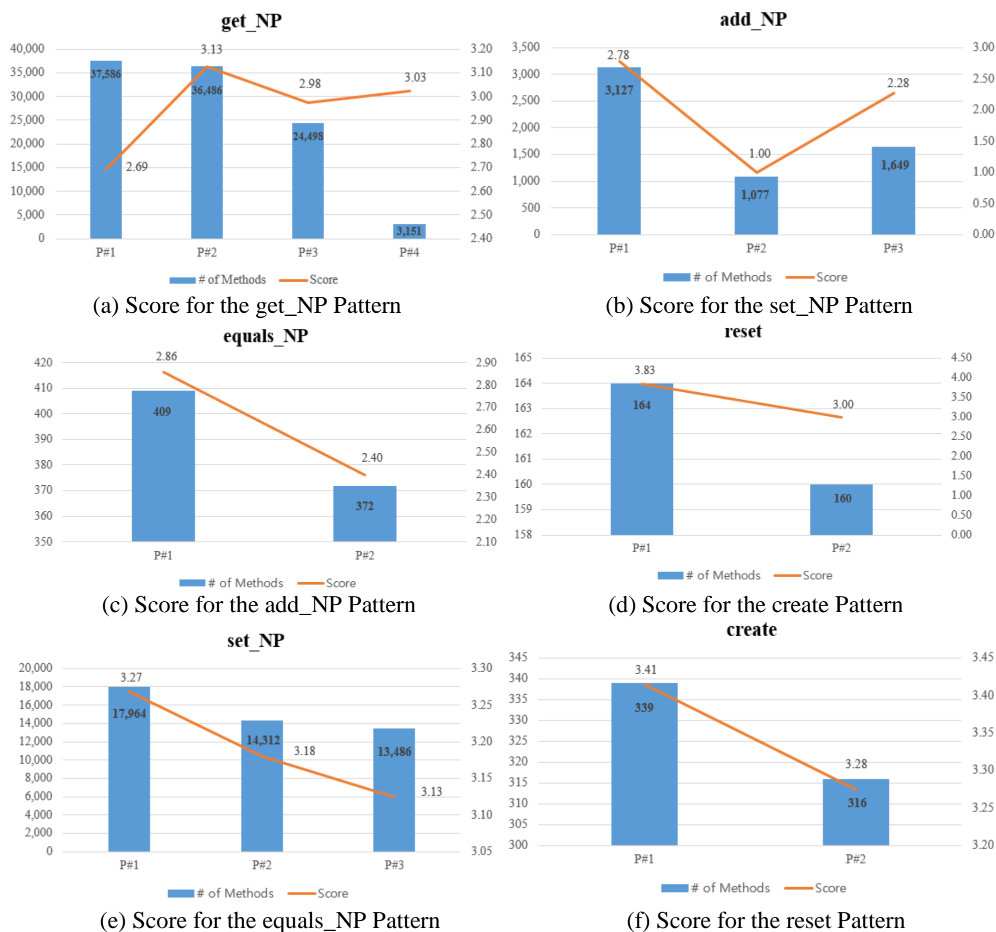
$$\text{Precision} = \frac{|\#\text{of } Appropriate\ Implementation\ Patterns|}{} \quad (4)$$

4.2.2. RQ2: *How different are statistics and human-validation of implementation patterns?*

The second research question is intended to figure out a gap between statistics and

human-validation. If there is a huge gap between statistics of our approach and human-validation, the identified implementation patterns can be thought invalid. **Fig. 7** shows statistics in the bar graph and human-validation of each pattern in the line graph at the same time to figure out the gaps. According to the figure, most of the statistics and human-validation have a similar score-frequency trend. For example, the *set_NP()* method identifier style has three implementation patterns, and the number of method identifiers of each pattern from our approach is in proportion to the average point of each pattern from human-validation. Even, the *add_NP()* style method identifier has the similar trend.

However, the pattern #1 and # 4 of the *get_NP()* style method identifier have different trends. The representative method code for the pattern #1 (see **Fig. 5(a)**) is the *getRandom()* method where the implementation only reflects the verb and objective part of the method identifier without considering the return type and parameters. Thus, the pattern obtained relatively lower point rather than its number of methods. Also, the pattern #4 got relative higher point compared to the number of methods, because the body part of the method *getTypeForFactoryMethod()* appropriately realizes the intention of method identifier as well as parameters and a return type. This experiment for RQ2 shows no huge gap between statistics and human-validation, which indicates that most of the method implementation patterns identified by our approach are considered appropriate from human perspective.



(a) Score for the get_NP Pattern

(b) Score for the set_NP Pattern

(c) Score for the add_NP Pattern

(d) Score for the create Pattern

(e) Score for the equals_NP Pattern

(f) Score for the reset Pattern

Fig. 7. The Scores of the Sample Methods

5. Conclusion

In this paper, we proposed an automatic approach to identifying Java method implementation patterns associated with method identifiers. Thus, we first suggested the three types of feature vectors to characterize the method signature part, body part and their relation. Then, we enhanced the K-medoids algorithm by applying the Calinski and Harabasz algorithm, and suggested Cascade K-medoids that automatically decides the optimal K, which is the number of implementation pattern. Based on the algorithm, we identified 16,768 implementation patterns of 7,169 method identifiers from 50 open source projects and obtained 86% of precisions. As future work, we have a plan that enhance our approach more by applying diverse deep learning technologies to increase the precision and recall of our experiment. In addition, we will develop tool support as Eclipse Plugin-In to automatically suggest implementation patterns appropriate to a method in typing the method signature on the fly.

Acknowledgement

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning(NRF-2014M3C4A7030503).
This research is supported by Ministry of Culture, Sports and Tourism (MCST) and Korea Creative Content Agency (KOCCA) in the Culture Technology (CT) Research & Development Program 2016 (R2016030046).

References

[1] K. Beck, “Implementation Patterns 1st Edition,” *Addison-Wesley Professional*, 2007. Article(CrossRefLink)
[2] Thomas M. Pigoski, “Practical Software Maintenance: Best Practices for Managing Your Software Investment,” *Wiley Publishing*, 1st edition, 1996. Article(CrossRefLink)
[3] Joseph (Yossi) Gil and Itay Maman, “Micro patterns in java code,” in *Proc. of Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA ’05*, pages 97–116, New York, NY, USA, 2005. ACM. Article(CrossRefLink)
[4] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami, “Mining association rules between sets of items in large databases.” *SIGMOD Rec.*, vol. 22, no. 2, 207–216, June 1993. Article(CrossRefLink)
[5] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden, “An approach for evaluating and suggesting method names using n-gram models,” in *Proc. of Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014*, pages 271–274, New York, NY, USA, 2014. ACM. Article(CrossRefLink)
[6] C. E. Shannon. “A mathematical theory of communication,” *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, no. 1, 3–55, January 2001. Article(CrossRefLink)
[7] F. Deißenbock and M. Pizka, “Concise and consistent naming,” in *Proc. of Proceedings of the International Workshop on Program Comprehension (IWPC’05)*, pages 97–106. IEEE CS Press, 2005. Article(CrossRefLink)
[8] D. Lawrie, H. Feild, and D. Binkley, “Syntactic identifier conciseness and consistency,” in *Proc. of Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, 27-29 September 2006, Philadelphia, Pennsylvania, USA, pages 139–148, 2006. Article(CrossRefLink)

[9] George A. Miller. "Wordnet: A lexical database for English," *Commun*. ACM, vol. 38, no. 11, 39–41, November 1995. Article(CrossRefLink)
[10] Eclipse. Eclipse Class ASTParser. Online: Article(CrossRefLink)
[11] Oracle. Code Conventions for the Java Programming Language: Why Have Code Conventions SunMicrosystems. Online: Article(CrossRefLink).
[12] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Empirical Software Engineering*, vol. 21, no.v2, 565–604, 2016. Article(CrossRefLink)
[13] Stanford. The Stanford Parser: A statistical parser Homepage. Online: Article(CrossRefLink).
[14] S. Kim, T. Kim, I. Lee, J.A Kim, and Y. Cho, "Feature vectors for recognizing java method naming patterns," in *Proc. of Asia Pacific International Conference on Information Science and Technology*, pages 320–322. IEEE, 2017.
[15] Hae-Sang Park and Chi-Hyuck Jun, "A simple and fast algorithm for k-medoids clustering," *Expert Systems with Applications*, vol. 36, no. 2, Part 2, 3336 – 3341, 2009. Article(CrossRefLink)
[16] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, 100–108, 1979. Article(CrossRefLink)
[17] T. Calinski and J. Harabasz, "A dendrite method for cluster analysis," *Communications in statistics*, vol. 3, no. 1, 1–27, 1974. Article(CrossRefLink)
[18] William B. Frakes and Ricardo Baeza-Yates, "Information Retrieval: Data Structures and Algorithms," *PrenticeHall, Inc.*, 1st edition, 1992. Article(CrossRefLink)
[19] I.H. Witten, E. Frank, and M.A. Hall, "Data Mining: Practical Machine Learning Tools and Techniques, Third Edition(Morgan Kaufmann Series in Data Management Systems)," *Morgan Kaufmann*, 2011. Article(CrossRefLink)

Appendix

Table 5. Open Source Project List for Training

Name	Version	Description	# of Methods
Ant	1.9.2	Build Tool	4,694
Antlr4	4.3.0	simple description	2,454
Asm	5.0.1	Another Tool for Language Recognition	4,712
Axis	1.4.0	SOAP Engine	10,040
Activemq	5.4.1	JMS Messaging Engine	4,651
ArgoUML	0.32.0	UML Modeling Tool	2,361
Apache Collections	4.1.0	Data Structure Framework	4,480
Apache poi	3.12.0	MS Office Manipulation Framework	3,785
Apache-commons math	3.4.1	Math Library	4,266
Apache-camel	2.17.7	Routing and mediation engine	1,251
Apache-commons io	2.0.1	IO Framework	1,942
Apache tomcat	8.5.16	Servlet Engine	5,811
Apache Commons Codec	1.5.0	Codec Framework	865
Apaache Commons Dbcp	2.0.1	Database Connection Pool Engine	2,066
Apache Derby	10.8.1.2	Relational Database	30,953
Apache Bsf	3.0.0	Script Language Support	1,238

Apache Lucene	6.0.1	Search Engine	6,487
Apache Jena	2.12.1	RDF/SPARQL Framework	2,331
Apache Log4j	2.3.0	Logging Framework	1,693
Accumulo	1.7.1	A key/value store for BigTable	2,625
Base	0.8.11	Baidu Search Marketing Service API	14,119
Beanshell	1.3.0	Lightweight Scripting for Java	15,429
c3po	0.9.5.2	JDBC DataSources/Resource Pools	4,119
Castor	1.1.2	XML framework	13,900
Cglib	2.2.2	Code Generation Library	1,709
Dom4j	1.5.1	XML Framework	2,521
googlewebtoolit	1.4.10	Google Web Toolkit	7,809
Hibernate-orm	5.1.6	ORM Framework	41,745
jacORB	3.2.0	Java Implementation of OMG's CORBA	11,850
javaassist	3.19.0	Java bytecode engineering toolkit	4,783
Jboss	6.1.0	Web Application Server	3,348
Jgroup	2.6.21	Group communication toolkit	5,609
Jedit	5.3.0	Text Editor	18,124
Jmeter	2.12.0	Stress Testing Tool	5,118
Jruby	9.0.5.0	Ruby programming language atop the JVM	16,104
Junit	4.8.2	Unit Testing Tool	2,706
Jxta	2.0.0	Java P2P Programming	7,026
Jython	2.5.3	Python designed to run on JVM	42,439
Kawa	2.0.0	A general-purpose programming language on JVM	8,554
Apache mailbox	0.5.0	Mailbox Storage Implementation	2,666
JavaCompiler	1.6.0	Java Compiler	47,821
modeler	1.1.0	JMX Support Library	431
MX4J	2.0.1	Open Source JMX for Enterprise Computing	4,009
OpenJMS	0.7.5	Open Source Implementation	2,660
Pico	3.9.5	A framework for Behaviour-Driven Development(BDD)	2,685
Pool	2.3.0	Object Pool Pattern Implementation	770
sandbox	6.3.0	Lucene Support Library	3,074
Spring-Framework	2.5.6	Framework for Java based Enterprise Applications	53,591
jackrabbit	2.14.1	Content Repository	5,124
JHotDraw	5.2.0	GUI Toolkit	4,685

Table 10. Implementation Patterns, Methods and Sores

Method Identifier	# of Methods	Im. Patterns	# of Methods	Questionnaire Average Score
get_NP	101721	#1	37,586	2.69
		#2	36,486	3.13
		#3	24,498	2.98
		#4	3,151	3.03
set_NP	45762	#1	17,964	3.27
		#2	14,312	3.23
		#3	13,486	3.14
get_NP_NP	5476	#1	2,146	3.57
		#2	1,831	3.01
		#3	1,229	2.34
add_NP	5853	#1	3,127	2.76
		#2	1,649	1.03
		#3	1,077	2.33
create_NP	5845	#1	3,754	3.32
		#2	2,091	2.37
file_A	2865	#1	876	3.41
		#2	753	2.73
		#3	685	3.07
		#4	551	3.38
is_NP	2192	#1	1,317	2.96
		#2	875	2.73
compare	1654	#1	832	3.31
		#2	822	3.29
find_NP	1551	#1	862	2.72
		#2	689	2.38
set_NP_NP	1428	#1	945	3.28
		#2	483	3.02
readLine	1374	#1	617	3.57
		#2	247	1.96
clone	1159	#1	678	3.32
		#2	481	1.98
init_NP	1158	#1	420	2.62
		#2	376	3.28
		#3	362	3.11
accept_NP	903	#1	472	2.43
		#2	431	1.95
cast_NP	876	#1	493	2.62
		#2	383	2.73
read_NP	864	#1	617	2.68
		#2	247	1.97
recreate_NP	784	#1	583	3.33
		#2	201	2.64
equals_NP	781	#1	409	2.88
		#2	372	2.42
close_NP	778	#1	413	2.94
		#2	365	1.96
convert_A	681	#1	348	1.95
		#2	333	2.67

remove_NP	669	#1	471	2.93
		#2	198	2.48
replace_NP	659	#1	343	3.62
		#2	316	3.29
discover_NP	659	#1	426	2.71
		#2	233	1.95
create	655	#1	339	3.42
		#2	316	3.25
draw_NP	481	#1	296	2.51
		#2	185	2.28
contain_NP	457	#1	234	3.13
		#2	223	2.57
isValid_NP	413	#1	267	2.86
		#2	146	1.87
compute_NP	364	#1	193	2.92
		#2	171	2.83
contains	358	#1	220	3.13
		#2	138	2.38
reset	324	#1	164	3.78
		#2	160	2.94

Tae-young Kim is currently in the master course in Department of Software Engineering at Chonbuk National University. He received the B.S. degree in the Department of Software Engineering at Chonbuk National University in 2016. His research focuses on source code mining, and machine learning.

Suntae Kim is an Associate Professor of the Department of Software Engineering at Chonbuk National University. He received his B.S. degree in computer science and engineering from Chung-Ang University in 2003, and the M.S. Degree and Ph.D. Degree in computer science and engineering from Sogang University in 2007 and 2010. He worked in Software Craft Co. Ltd., as a senior consultant and engineer for financial enterprise systems during 2002, 2004. Also, he developed Android based Smart TV middleware from 2009 to 2010. His research focuses on software architecture, design patterns, requirements engineering, and source code mining.

Jeong Ah Kim, received Ph. D degree at ChungAng University. Since 1996, she has worked at Catholic Kwandong University as professor. She is the member of Korea Institute of Information Science and Engineering and the member of board of directors of Convergent Research Society. Her research areas are software product line engineering, software modeling, software process improvement, clinical decision support system.

Jae-Young Choi is a professor with the department of computer engineering, college of software at the Sungkyunkwan University, Korea. He received his B.S. degree in mathematics in 1995, and the M.S. and Ph.D. degrees in computer science from the Kyungwon University, Korea, in 1999 and 2004, respectively. From 2004 to the middle of 2006, he joined the Vision Laboratory at the University of California, Los Angeles, USA, as a postdoctoral researcher. He has also served as a BK21 research professor at Kyungwon University from 2006 to 2010. His research interests include computer vision, machine learning, ubiquitous computing, network management, software engineering and R&D strategies.

Jee-Hyong Lee received his B.S., M.S., and Ph.D. in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Rep. of Korea, in 1993, 1995, and 1999, respectively. From 2000 to 2002, he was an international fellow at SRI International, USA. He joined Sungkyunkwan University, Suwon, Korea, as a faculty member in 2002. His research interests include fuzzy theory and application, intelligent systems, and machine learning.

Young-Hwa Cho is currently a distinguished visiting professor in the college of software at the Sungkyunkwan University, Korea. He received his B.S. degree in statistics and the M.S. degree in computer science from Sungkyunkwan University in 1977 and 1990, respectively. In 1999, he completed his Ph.D. degree in computer science from Chungbuk University, Korea. He was with KISTI(Korea Institute of Science and Technology Information) as a President from 2001 to 2006. He has also served as a President at KISTEP(Korea Institute of S&T Evaluation and Planning) from 2007 to 2008. He joined Kyungwon University as a visiting professor from 2008 to 2010. His current research interests cover software engineering, data base, information communication technology, R&D strategies and so on.

Young-Kwang Nam is a professor with the department of Computer and Telecommunications from Yonsei University. He received his B.S. degree in mathematics from Yonsei University in 1982, and the M.S. and Ph.D. degrees in computer science from the KAIST and Northwestern University, in 1985, 1991, respectively. He was a Senior Researcher at SERI(System Engineering Research Institute). His research areas are Programming Language, Software Engineering, Information Retrieval, Database, XML.