# A Metric-based Approach for Reconstructing Methods in Object-Oriented Systems

Tatsuya Miyake Yoshiki Higo Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
{t-miyake,higo,inoue}@ist.osaka-u.ac.jp

## ABSTRACT

Refactoring is an important activity to improve software quality, which tends to become worse through repetitive bug fixes and function additions. Unfortunately, it is difficult to perform appropriate refactorings because a refactoring needs certain costs, and its effects should be greater than the costs. This paper describes an approach for appropriate refactorings. The approach identifies spots to be refactored and it suggests how they should be improved. Moreover, the approach estimates the effects of the refactorings. The approach requires a lightweight source code analysis for measuring several metrics, so that it can be applied to middle- or large-scale software system. The approach can make the refactoring process more effective and efficient one.

## Keywords

Refactoring, Software maintenance, Source code analysis

## 1. INTRODUCTION

Quality of a software system is a key factor whether its development and maintenance succeed or not. Regrettably, software quality grows worse by project creep, repetitive bug fixes, function additions and so on. It is not realistic that quality of a software system doesn't become worse through its life cycle.

Refactoring is an activity to improve the internal structure of a software system without changing the external behavior of it [5], and it can regain software quality. A refactoring itself requires certain costs, and the effects should be greater than the costs. But the effects is ambiguous before performing it, so that we can say that it is difficult to perform appropriate refactorings.

In this paper, we propose an approach for performing appropriate refactorings. The approach focuses on the internal structures of methods in object-oriented systems. Firstly, the approach identifies spots to be refactored, and then it suggests how they should be improved. Moreover, the approach estimates the effects of the refactorings that it suggested. The approach is metric-based, and it requires a lightweight source code analysis, so that it can be applied middle- or large-scale software systems.

As of now, we are implementing a software tool which realizes the approach. This paper describes the details of the approach and a case study with the partially implemented tool. After The software tool is completed, we will do more case studies.

## 2. REFACTORING

Refactoring is a set of operations to improve maintainability, understandability, extendability or other attributes of software systems without changing the external behavior of it [5], and it is getting much attention recently. Usually, refactoring is performed on the following process [7].

**STEP1** : Identifies spots that should be bottlenecks of development or maintenance of the software system. For example, too long methods, complicated control structures, or duplicated code should be refactored [5].

**STEP2** : Determines how the spots should be improved. Before now, many refactoring patterns have been proposed [2, 5].

**STEP3** : Estimates the costs and the effects of the refactoring determined in STEP2. If the effects will be greater than the costs, the refactoring should be really performed.

**STEP4** : Changes the source code based on the refactoring determined in STEP2. Recently, IDEs like Eclipse [4] or NetBeans [1] have functions changing source code automatically.

**STEP5** : Conducts regression test to ensure that the refactoring didn't change the external behavior of the software system.

A problem of refactoring is that STEP1 and STEP2 need rich knowledge and experiences to identify spots to be refactored and to determine how the spots are improved. In practice, there are many developers having limited knowledge and low experiences, so that they spend too much time in STEP1 and STEP2. Another problem is that it is difficult to estimates the effects of a refactoring before performing it. Developers don't tend to be aware that the refactoring has side-effects before they actually perform it. Thus inappropriate refactorings often may be performed.

In this paper, we propose an approach to support STEP1, STEP2, and STEP4 of the refactoring process. By using the approach, the users can perform appropriate refactorings efficiently.

**Table 1: Attribute encapsulation and local variable capsulation**

| Name | Attributes encapsulation | Local variable encapsulation |
|---|---|---|
| Summary | Encapsulates attributes for preventing them from being accessed from other classes. | Encapsulates local variables for preventing them from being accessed from other spots in the method. |
| Implementation | Attributes are defined as private. If necessary, accessors of them are defined. | A spot of a method is extracted as a new method. If necessary, argument and return variable are used. |

# 3. APPROACH

This section describes the proposed approach. In the approach, Refactoring means that *structural blocks in methods of object-oriented systems are extracted as new methods in the same class or another class*. For example, in the case of Java language, there are 8 types of strucutral blocks (do, if, for, switch, synchronized, simple-block ({}), try, while).

Section 3.1 describes how the method identifies spots to be improved, and Section 3.2 mentions how the spots to be refactored. Finally, Section 3.3 presents the effects estimation of the approach.

## 3.1 Identifies spots to be improved

The proposed approach investigates the internal structures of the methods of a software system, or several metrics are measured on structural blocks in the methods. Blocks having bad metrics values are the targets to be refactored. The following describes metrics used in the approach.

**Cyclomatic complexity** : Cyclomatic complexity is the number of linearly independent paths from the start node to the end one of a graph [6]. In the case of source code, Cyclomatic complexity can be represented by the number of conditional expressions plus 1. It is widely accepted that high value of cyclomatic complexity of a method implies that it includes complicated control logics and it is hard to understand and to conduct enough tests. The proposed approach measures cyclomatic complexity of structural blocks in addition to methods in the target software system.

We assume that there are $n$ expressions in block $B$, metric $CC(B)$ can be represented as follows.

$$CC(B)(CyclomaticComplexity) = n + 1$$

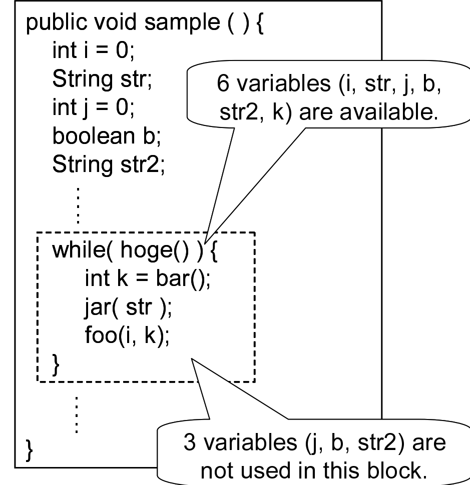
**LOC (Lines Of Code)** : LOC is the simplest and the most widely-used metric to measure module size in a software system. The proposed approach measures LOC of structral blocks and methods of the target software system.

Additionally, in the case of a structural block, the approach calculates its occupancy rate for the method including it. The occupancy rate of a structural block can be a indicator whether the structural block should be refactored or not.

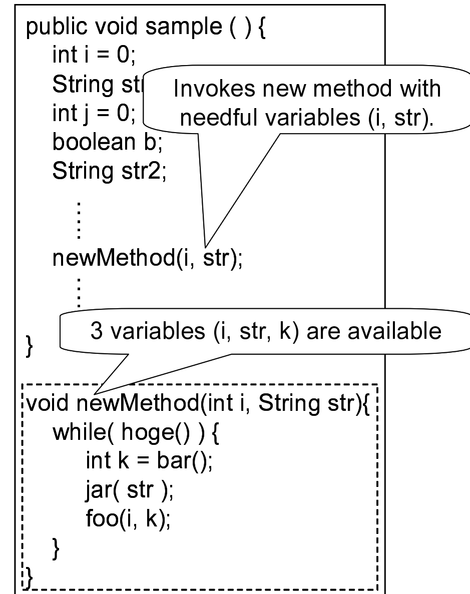
We assume that the LOC of block $B$ is $LOC(B)$ and the LOC of method $M$ including the block is $LOC(M)$, metric $OR(B)$ can be represented as follows.

$$OR(B)(OccupancyRate) = \frac{LOC(B)}{LOC(M)}$$

For example, there are two methods, $A$ and $B$: method $A$ consists of 100 lines of code and it includes a 50 LOC block; method $B$ consists of 60 lines of code and it includes a 50




(a) Before refactoring



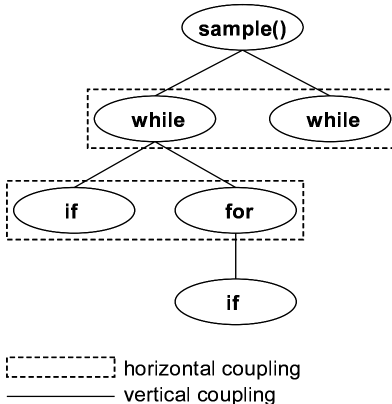

(b) After refactoring

**Figure 1: Refactoring example reduce the number of available local variables**

LOC block; the both blocks have 50 LOC whereas the occupancy rate of $A$ is $\frac{50}{100} = 0.5$, and $B$ is $\frac{50}{60} = 0.83$; in the case of $A$, by extracting the block as a new method, $A$ is divided into two reasonable size methods; in the case of $B$, extracting the block means almost all of code of $B$ is moved to the new

```
public void sample ( ) {
    int i = 0;
    String str;
    while( hoge() ) {
        str  = "string";
        if( jar( I ) ) {
            make(str);
        }
        foo(i);
        for( int j = 0; j < I ; j++) {
            sam( i );
            if( log ) {
                System.out.println(i);
            }
        }
    }
    ⋮
    while( ) {
        bar(i);
        make(str);
    }
}
```

(a) Source code of a method




(b) Tree structure of the method

**Figure 2: Horizontal Coupling and vertical Coupling**

Before refactoring

```
public void sample ( ) {
    ⋮
    while( hoge() ) {
        int i = 0;
        String str  = "string";
        foo(i);
    }
    ⋮
}
```

After refactoring

```
public void sample ( ) {
    ⋮
    newMethod();
    ⋮
}

private void newMethod ( ) {
    while( hoge() ) {
        int i = 0;
        String str  = "string";
        foo(i);
    }
}
```



(a) Week coupling

Before refactoring

```
public void sample ( ) {
    int i = 0;
    boolean bool ;
    String str;
    ⋮
    while( hoge() ) {
        str  = "string";
        foo( i );
        bar( bool );
    }
    ⋮
    return str;
}
```

After refactoring

```
public void sample ( ) {
    int i = 0;
    boolean bool ;
    String str;
    ⋮
    str = newMethod(i, bool);
    ⋮
    return str;
}

private String newMethod
            (int i, boolean bool) {
    String str;
    while( hoge() ) {
        str  = "string";
        foo( i );
        bar( bool );
    }
    return str;
}
```



(b) Strong coupling

**Figure 3: Extraction based on vertical coupling**

method. The refactoring of $B$ should not be appropriate.

**Number of Available Variables** : In object oriented systems, attributes of classes shouldn't be able to access outside the class. If we want to uses an attribute of a class in another class, we use accessor method for the attribute. Attribute encapsulation can reduce the number of available attributes, and it can make the source code more robust.

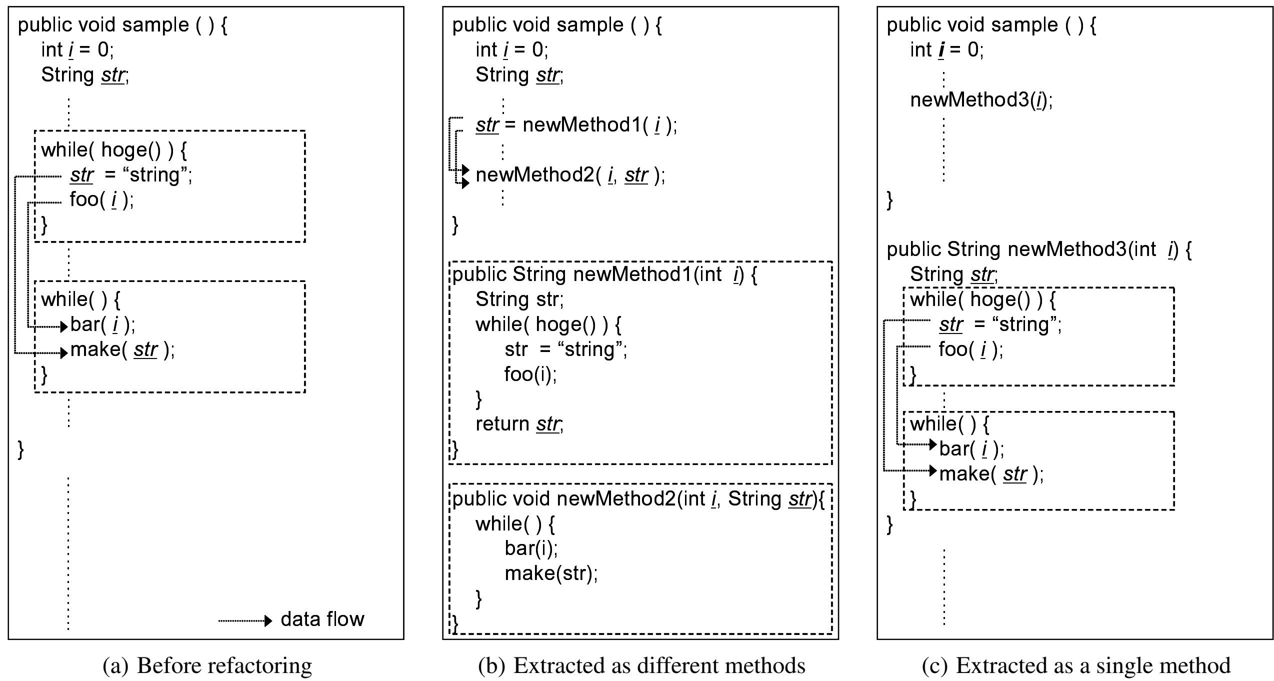
Here, we propose *local variable encapsulation*, which is the same concept as attribute encapsulation. If we encapsulate local variables that are not used in some spots of the method, the maintainability of the method should be improved. Local variable encapsulation can prevents encapsulated local variables from being accessed accidentally by changing the source code. Table 1 illustrates a summary of attribute encapsulation and local variable encapsulation.

We defined a new metric $ALV(B)$ as follows.

$ALV(B)$ (Available Local Variable) is the number of local variables that are defined outside block $B$ and that can be accessed in the block.

We think that, spots to be refactored have higher value of the number of available local variables than spots not to have to be refactored. Figure 1 illustrates a sample refactoring reducing the number of available local variables. Before refactoring, there are 6 available local variables (i, str, j, b, str2, k) in the hatching part. But 3 local variables (j, b, str2) of them are not used in the part. If the part were changed because of bug fix or function addition, these 3 variables might be used carelessly and some new bugs may be yielded. In this situation, extracting the part as a new method like Figure 1(b) can prevents these local variables from being accessed accidentally.

The metrics described above are used to identify spots to be improved. Because this study is in progress, we have not known reasonable thresholds of these metrics yet.




(a) Before refactoring (b) Extracted as different methods (c) Extracted as a single method

**Figure 4: Refactoring example based on horizontal coupling**

## 3.2 Determines how the spots to be refactored

In this process, couplings of blocks with their surrounding code are measured. The coupling metrics are used to determine how to refactor the blocks.

**Vertical coupling** : Vertical coupling means a coupling of a block with its outer block. This coupling is measured based on how the block uses local variables that are defined outside it. Henceforth, we call such variables *outer variables*.

When a block is extracted as a new method,

- if a block refers to outer variables, their values have to be passed to the new method as arguments,
- if a block assigns some values to outer variables, the new methods returns the variables as return value.

Figure 3 presents two examples: one is a case that the target block has week coupling with its outer block; the other is a case that the target block has strong coupling. In the case of Figure 3(a), the target block uses no outer variable, so that the block can be extracted as a new method simply. On the other hand, in the case of Figure 3(b), the target block refers to and assigns a value to outer variables, so that extracting the block requires to add arguments and a return statement to the new method.

Two metrics $NRV(B)$ and $NAV(B)$ can be defined as follows.

$NRV(B)$ (Number of Referred Variables) of block $B$ is a number of outer variables that are referred to in the block.

$NAV(B)$ (Number of Assigned Variables) of block $B$ is a number of outer variables that are assigned to in the block.

**Horizontal coupling** : Horizontal coupling means a coupling of two blocks in the same scope. If the coupling is strong, the blocks should be extracted as a new method. Otherwise the blocks should be extracted as different new methods. The metric $HC(B_s, B_t)$ can be defined as follows.

$HC(B_s, B_t)$ (Horizontal Coupling) is a number of outer variables used (referred or assigned) in both block $B_s$ and $B_t$.

Figure 4 represents a sample refactoring based on horizontal coupling. Before refactoring (Figure 4(a)), there are two blocks in the method and these blocks have a certain horizontal coupling because there are two data flows between them. If these blocks are extracted as different methods (Figure 4(b)), they have to have complicated signature (arguments and return value) because the data have to be passed through the new methods. On the other hand, if the blocks are extracted as a single method (Figure 4(c)), the signature of the new method become simple because the data flows are completed within the new method.

But, from the view point of the modularity, Figure 4(b) may be better than Figure 4(c) because each functionality is divided into different methods. This is very sensitive problem, so that we cannot say which is the best solution. However, we believe that the horizontal metric should be a indicator how blocks should be refactored.

In the current definition of metric $HC$, we don't differentiate assignment from reference. For more precisely representing how blocks can be refactored, it is vital to distinguish between assignment and reference. Moreover, other techniques of source code analysis like program slicing should be used. In the current definition of horizontal coupling, the following coupling cannot be grasped.

**Table 2: The target systems**

| Name | Version | # Files | LOC | Analysis Time |
|---|---|---|---|---|
| JHotDraw | 5.4b2 | 289 | 40,986 | 3.7 secs |
| Ant | 1.6.5 | 674 | 166,295 | 12.6 secs |
| Antlr | 3.0.1 | 142 | 44,032 | 4.1 secs |

- Firstly, in block $B_1$, a value is assigned to variable $V_1$.
- Then, a value is calculated using variable $V_1$, and it is assigned to variable $V_2$ (these operation are performed outside $B_1$).
- Finally, in the block $B_2$[1], variable $V_2$ is referred to.

Moreover, in this process, the approach suggests where the refactored blocks should moved to. The suggestion is based on attribute usages and method invocations in the target blocks.

- If the blocks mainly use attributes and methods of its own class, the blocks are extracted in the same class.
- If the blocks use members of only its super class, the blocks are extracted and moved to the super class.
- If the blocks use members of other classes, the blocks are extracted and moved to the class whose members are used by the blocks or utility class.

## 3.3 Estimates the effects of the refactoring

The method estimates the effects of the refactoring by using metrics described in Section 3.1 and 3.2. The metrics values after refactoring can be estimated from the source code of before refactoring and refactoring operations. The estimated effects can be used for checking whether the refactoring lead to side-effects or not.

# 4. CASE STUDY

We have conducted simple case studies on open source software systems with the partially implemented tool. The target systems are JHotDraw, Ant, and Antlr, which are implemented in Java languages. Table 2 represents the names, the versions, the numbers of source files, and LOCs of the target software systems.

As you can see in the table, the tool could complete the source code analysis at short times despite the targets are middle-scale software systems[2]. The current implementation doesn't include program slicing. If we implement a program slicing function in the tool, the detection speed becomes much lower. Applying program slicing is one of the future works. Section 5.1 describes about it.

For the reason of the partial implementation of the tool, we cannot conduct a fully quantitative evaluation of the proposed approach at present time. In the following of this section describes two case examples detected in Ant and Antlr. We would like you to understand that we cannot describes the results of vertical coupling due to limitations of space.

[1]Blocks $B_1$ and $B_2$ are in the same scope in a method.
[2]The tool was executed on the PC workstation with 3.00 GHz CPU and 2.00 GB Memory.

```
00:private void parsePackages(Vector pn, Path sp) {
01:    Vector addedPackages = new Vector();
02:    Vector dirSets = (Vector) packageSets.clone();
       ........
08:    if (sourcePath != null && packageNames.size() > 0) {      [if1]
          ........
32:       for (int i = 0; i < pathElements.length; i++) {
             ........
37:          dirSets.addElement(ds);
38:       }
39:    }
       ...........
41:    Enumeration e = dirSets.elements();
42:    while (e.hasMoreElements()) {                             [while1]
          ........
49:       for (int i = 0; i < dirs.length; i++) {
             ........
60:          if (files.length > 0) {
                ........
64:             if (!addedPackages.contains(packageName)) {
65:                addedPackages.addElement(packageName);
66:                pn.addElement(packageName);
67:             }
68:          }
69:       }
70:       if (containsPackages) {
             ........
73:          sp.createPathElement().setLocation(baseDir);
74:       } else {
             .......
77:       }
78:    }
79:}
```

(a) A case of week horizontal coupling in Ant

```
00:protected void extractAttribute(String decl) {
        ........
04:     boolean inID = false;
05:     int start = -1;
06:     int rightEdgeOfDeclarator = decl.length()-1;
07:     int equalsIndex = decl.indexOf('=');
08:     if ( equalsIndex>0 ) {                                    [if1]
           ........
11:        rightEdgeOfDeclarator = equalsIndex-1;
12:     }
        ........
14:     for (int i=rightEdgeOfDeclarator; i>=0; i--) {            [for1]
15:        if ( !inID && Character.isLetterOrDigit(decl.charAt(i)) ) {
16:           inID = true;
17:        } else if ( inID && !(Character.isLetterOr ........
              ........
22:           start = i+1;
23:           break;
24:        }
25:     }
26:     if ( start<0 && inID ) {                                  [if2]
           ........
28:     }
        ........
34:     for (int i=start; i<=rightEdgeOfDeclarator; i++) {        [for2]
           ........
42:        if ( i==rightEdgeOfDeclarator ) {
43:           stop = i+1;
44:        }
45:     }
        ........
60:}
```

(b) A case of strong horizontal coupling in Antlr

**Figure 5: Examples idenfified in the target systems**

## 4.1 A case of week horizontal coupling

Figure 5(a) illustrates a case example of week horizontal coupling identified in Ant. The method in the figure includes two blocks, one is if-block (if1) and the other is while-block (while1).

In (if1), an externally defined variable *dirSets* is referred to once. On the other hand, in (while1), 4 externally defined variables *e*, *addedPackages*, *pn*, and *sp* are referred to. Thus, The coupling between (if1) and (while1) is week, they can be extracted as different methods simply. By extracting (if1) as a new method, three variables *pn*, *sp*, and *addedPackage* become invisible in it. The source code will become more robust because unnecessary information are hidden.

From the view point of size, the method before refactoring has 79 LOC, whereas all of refactored methods have about 10, 30, and 40 LOC. The refactoring divided the long method into three reasonable-size methods.

## 4.2 A case of strong horizontal coupling

Figure 5(b) represents a case example of strong horizontal coupling identified in Antlr. The method in the figure includes four blocks, two of them are if-blocks, (if1) and (if2), and the others are for-blocks, (for1) and (for2).

In (if1), a value is assigned to variable *rightEdgeOfDeclator*, and the variable is referred to in (for1) and (for2). It is possible to extract the three blocks as different methods with adding a return statement to the method of (if1).

In (for1), variables *intID* and *start* are assigned values to, and the both variables are referred in (if2). Thus, it is difficult to extract them as different methods, they has a strong horizontal coupling. For extracting them as different methods, we have to prepare a class that can store the both data. In this case, it is realistic that (for1) and (if2) are extracted as a single method with adding an argument for *rightEdgeOfDeclarator* to the method.

# 5. FUTURE WORK

## 5.1 Applying Program Slicing

The approach described in this paper is work-in-progress and we have many things to sophisticate it. The biggest problem is that, we think, the approach cannot identify statements related with the specified block. In the case of Figure 5(a), the statement in line 41 are related with both (if1) and (while1). The statement refers to variable *DirSets* which is assigned to in (if1), and it assigns a value to variable *e* which is referred to in (while1).

We are going to apply program slicing technique to identify such statements. Program slicing is performed from the externally defined variables in the specified block. The program slicing can identify statements related with the block. Figure 6 illustrates how (if1) can be extracted with the current approach and the approach with program slicing. The example is an actual source code identified in Ant, which is also represented as Figure 5(a). In the current approach (Figure 6(a)), the definition of variable *dirSets* exists in method *parsePackage* after extracting (if1). That means the variable can be accessed in the method, and it may be used in the method accidentally. In the future approach (Figure 6(b)), the definition of variable *DirSets* is moved to the extracted method. Thus, the variable cannot be accessed in method *parsePachages*, which can prevent the variable from begin used in the method accidentally.

```
00:private void parsePackages(Vector pn, Path sp) {
01:    Vector addedPackages = new Vector();
02:    Vector dirSets = getDirSets();
40:
41:    Enumeration e = dirSets.elements();
42:    while (e.hasMoreElements()) {          (while1)
          ........
78:    }
79:}
```

(a) Current Extraction

```
00:private void parsePackages(Vector pn, Path sp) {
01:    Vector addedPackages = new Vector();
40:
41:    Enumeration e = getDirSetsEnumeration();
42:    while (e.hasMoreElements()) {          (while1)
          ........
78:    }
79:}
```

(b) Future Extraction

**Figure 6: Examples of Current and Future Extraction**

## 5.2 Correlation with other metrics

In this paper, we proposed a new metric *ALV* (Available Local Variable), which should be a indicator whether or not the block should be refactored or not. We have to evaluate whether the metrics have a correlation with the number of bugs or other metrics. We are going to evaluate correlations with CK metrics suite [3].

## 5.3 Interview

In addition to quantitative evaluation described in Section 5.3, we are going to conduct interviews with developers of target software systems. Interview will provide us information that cannot be provided from the quantitative evaluation. The targets are both commercial and open source systems, and we will get different comments from the developers.

# 6. CONCLUSION

In this paper, we described an approach to reconstruct methods in object oriented systems. The approach performs source code analysis for (1)identifying spots to be improved, (2)suggesting how the spots to be improved, and (3)estimating the effects of the refactorings. this information is provided to the users automatically. The users determines whether the refactoring should be performed or not by themselves. At the end, we have to say that the approach is work-in-progress and we have many future works.

# 7. REFERENCES


[1] NetBeans. http://www.netbeans.org
[2] Refactoring home page. http://www.refactoring.com/
[3] S. Chidamber and C. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 25(5):476–493, Jun 1994.
[4] Eclipse. http://www.eclipse.org/.
[5] M. Fowlor. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.
[6] T. Macabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec 1976.
[7] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb 2004.