Journal of Engineering Technology

Volume 7, Special Issue (Internet of Things),Oct. 2018, PP. 262-272

RESEARCH ARTICLE -ENGINEERNG TECHNOLOGY ASEE

Enhancing Reusability with Reusable Code Patterns based on Reverse Engineering

Eun Young Byun[1], Byungkook Jeon* and R. Young Chul Kim[2]

[1,2] SE Lab., Hongik University, Sejong Campus, 30016, Korea
*Dept. of Software, Gangneung-Wonju National University, Wonju City, Gangwon Prov. 26403, Korea

Abstract: For quantitatively measuring software quality, there are existing some factors: complexity, reusability, performance and maintainability. Reusability of these factors is an important issue to improve productivity and quality at rapid software development. Current remarkable software metrics have some problems which are inadequately reflecting object oriented mechanism and unclearly dealing with reusability. To solve this problem, we extend with the previous metrics to reflect the OO mechanism such as hierarch, diversity, complexity, etc. To automate the measurement and identify the reusability, we develop a tool chain to extract static/dynamic reusable modules based on reverse engineering. We can automatically visualize to identify static & dynamic reusable modules from source code on the basis of our object oriented reusable metrics (OORM). To verify OORM and the tool chain, we visualize and compare JUnit3.4 and JUnit4.0. This may be expected to improve the productivity of new system with the reusable modules of legacy system.

Keywords: Reusability, Object Oriented Reusability Metrics (OORM), Reusable module

1 Introduction

Software metrics have been researched for quantifying software. By these metrics, 1) Project managers will know when the software is prepared to release, and whether the project is over budget, 2) Developers will verify the quality of software. 3) Maintainers will figure out whether the system has to be updated and improved by evaluating the system [1-9]. Nevertheless, it is very difficult to develop high-quality software. In addition, until now, we find a problem that object oriented paradigm does not completely reflect in many metrics. The existing metrics were based on procedural paradigm. Therefore, applying them to object oriented programs can lead to inaccurate meaning views.

In the software market, reuse is prevalent with APIs, open source, and library. If the develop will minimize his effort with the reuse of high quality reusable components, we can reduce the cost and time of a project. However, the issue of reuse is very important one for improving productivity. Object oriented software is difficult to modularize code with its invisible characteristics, which becomes more considered due to the interrelationships between them such as classes, objects, and class-object relations.

In reuse, there are important issues of "Which module should be reused?" as follows:

1) If the quality of the reuse module is low, the quality of the program may be deteriorated.

2) If a reuse module has strong coupling with other modules, this module may cause potential errors.

3) If the reuse module is unused, the code is the dead code.

To solve this problems, we analyze problems of existing metrics, and define object oriented reusability metrics improved with CK metrics and Cyclomatic metrics. We develop a visualization system, that is, a tool chain to measure these metrics, which can automatically identify reusable modules. With this approach, we can visualize the inner structure of code. This helps the understanding of a program to developers, and also guidelines for reuse.

The chapter 2 describes software visualization and problems of the previous reusability metrics in related work. The chapter 3 defines object oriented reusability metrics (OORM). The chapter 4 explains the automatic reusable code pattern identification based on OORM, and shows how to construct the visualization system. Finally, we mention conclusions and future work.

2 Related Work

Our previous visualization researches were conducted [4-5]. By visualizing the internal structure of the software with invisible characteristics, we improved the programmer's understanding of a source program, and measured the complexity of the code with various factors such as cohesion, coupling, bad smell, and performance. The code complexity can lead to some potential errors, resulting in lower productivity. Although inevitable complexity cannot be reduced, we must reduce unnecessary complexity. In this paper, we are focusing on reuse through code visualization [3-5]. Many researches have been done on metrics for measuring software. Typical metrics include CK metrics, Cyclomatic complexity, and so on. The CK metrics was defined in terms of complexity, reusability, encapsulation, and modularity on object oriented paradigm [6-11]. Kevin [12] mentioned the features that made software easier to reuse were called modularity with low coupling, high cohesion, and encapsulation. According to this, reusability is on the basis of modularity. These factors are the same as those affecting complexity. According to Marko [6], the problems of existing metrics can be derived as follows:

1) They are biased toward static factors, but dynamic factors is very lack.
2) The CK metrics do not consider the coupling and the cohesion which are typically used for reusability measurement.
3) In coupling one, only the call coupling is considered but, not the return coupling. Therefore, it is impossible to identify only with calling relations.
4) The Cyclomatic complexity has limitations in measuring the complexity of object oriented program.
5) The most important feature of object oriented mechanism is inheritance. The inheritance improves internal reusability, maintenance efficiency, and deduplication. However, in reuse of legacy systems, this relationship must also be considered. Existing metrics take inheritance into account, but do not consider an overriding to lower inheritance. If the inheritance method is invoked using the super() method even where the overriding is done, the inheritance is maintained.
6) Due to polymorphism, dynamic binding can be identified only on dynamic analysis. In addition to polymorphism, dynamic binding of classes and objects should also be considered.

3 Object Oriented Reusability Metrics

We define to improve object oriented metrics which are focused on method, class, object, class-object, and component. The proposed metrics is divided into a static aspect that analyzes the source code without actual execution, and a dynamic aspect of the execution of the program.

3.1 Static Metrics

Method Level.

▪ Coupling Between Method (CBM): The extension of Coupling Between Object Classes (CBO), which is the number of calls between classes at the class level

▪ Method Coupling (MCP): classify the data/stamp/external/common/content coupling that is not distinguished from previous works. It has a sequential score of 1-6 from the data to the content coupling. The coupling is based on the inter-module invocations. Since multiple invocations are possible, the coupling of all invocation is summed. Also, since there is a return in every invocation, the coupling of the return data type is also considered. The formulas are as follows:

- Invocation Count (IC): The number of all invocation between methods.
- Coupling-In (CP-In): The degree to which it is coupled by the invocation.

$$\mathrm{CP-In} = \sum_{i=1}^{IC} CP - In_i \quad (1)$$

- Coupling-Out (CP-Out): The degree to which it is coupled by the return.

$$\mathrm{CP-Out} = \sum_{i=1}^{IC} CP - \mathrm{Out}_i \quad (2)$$

- Method Coupling (MCP): The coupling between methods.

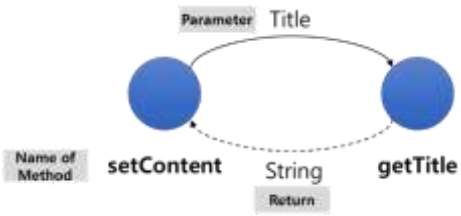
$$\mathrm{MCP} = \mathrm{CP-In} + \mathrm{CP-Out} \quad (3)$$

Table 1 is a sample code for MCP. The structure of a code consists of the method node, the call edge, and the return edge. A Class ‘Document’ consists of the method ‘setContent’ and method ‘getTitle’. There is a single invocation between these methods. Both the invocation coupling and return coupling have a value of ‘2’ in the stamp coupling. Therefore, a MCP between this methods is ‘4’.

Table 1 A MCP Sample Code

Source Code	class Document {
	public void setContent(Content c) {
	setTitle(c.title);
	setHeading(c.heading);
	}
	public String setTitle(Title title) {
	return “Title : “ + title;
	}
	public String setHeading(Heading heading) {
	return “Heading : “ + heading;
	}
	}

Notation	

Structure	
Invocation Count(IC)	1
CP-In	$\sum_{i=1}^{1} CP - In_i = Stmap = 2$
CP-Out	$\sum_{i=1}^{1} CP - Out_i = Stmap = 2$
MCP	$\mathrm{CP-In} + \mathrm{CP-Out} = 2 + 2 = 4$

▪ Lack of Cohesion of Statement (LCOS): the extension of Lack of Cohesion of Methods (LCOM), which is an accessibility to global variables at the class level, to the method level. The formulas are as follows:

- Reference-In Count (RIC): the number of references to internal attributes of a method
- Reference-Out Count (ROC): the number of references to external attributes of a method
- Lack of Cohesion of Statement (LCOS): the percentage of external references on all attribute references in a method.

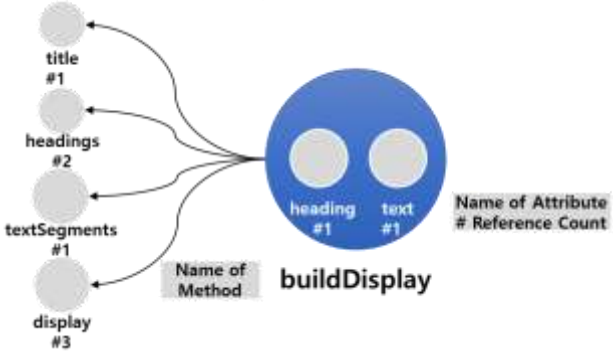
$$\mathrm{LCOS} = \frac{ROC}{RIC + ROC} \quad (4)$$

▪ Method Cohesion (MCH): classify the Coincidental / Logical / Temporal / Procedural / Communicational/Functional cohesion that are not distinguished from previous works. We refer to the definition of cohesion in [4], which redefine the cohesion of the existing procedural paradigm into an object oriented paradigm. The cohesion is referred to the degree to which the internal functions of a specific module are coupled. So this is only one value.
Table 2 is a sample code for LCOS. The structure of a code consists of the method node, the attribute node, and the reference edge. A Method 'buildDisplay' refers to two Attributes 'heading' and 'text' in the method, and other Attributes "title', 'headings', 'textsegments' and 'display' outside the method. About Method 'buildDisplay', RIC is '2' and ROC is '7'. As a result, LOCS is '0.77'.

Table 2 A LCOS Sample Code

Source Code	private String title;
	private Vector headings = new Vector();
	private Vector textSegments = new Vector();
	private Vector display = new Vector();
	
	public void buildDisplay() {
	display.addElement(getATitle(title));
	
	for(int i=0;i<headings.size();++i) {
	String heading = (String)headings.elementAt(i);
	display.addElement(getAHeading(heading, i+1));

	String text = (String)textSegments.elementAt(i);
	display.addElement(new Text(text));
	}
	}

Notation	
Structure	
RIC	2
ROC	7
LCOS	$\frac{ROC}{RIC + ROC} = \frac{7}{2+7} = 0.77$

Class Level.

The class level metrics is extended based on MCO as the method level metric. We define the class cohesion as the coupling of internal methods and the class coupling as a coupling between internal and external methods. The formula is as follows:

▪ Class Coupling (CCP): this is the sum of the MCOs of methods inside a class with external methods. The formula is as follow:

- Invocation-In Count (IIC): the number of the invocations between methods in a class.
- Class Coupling (CCP): the summation of the MCPs between internal methods in a class.

$$\text{CCP} = \sum_{i=0}^{IIC} MCP_i \quad (5)$$

▪ Class Cohesion (CCH): this is the sum of the MCOs between methods inside a class. The formula is as follow:

- Invocation-Out Count (IOC): the number of the invocations between internal methods and external methods.
- Class Cohesion (CCH): the sum of the MCPs between internal methods and external methods.

$$\text{CCH} = \sum_{i=0}^{IOC} MCP_i \quad (6)$$

Table 3 is a sample code for CCP and CCH. The structure of a code consists of the class node, the method node, the call edge, and the return edge. Class 'MailGenerationApplication' consists of the

method 'main' and method 'getCustomerTypeFromUser'. There are coupled with them. As a result, CCH is '3'. Also, there are coupled with the method 'sendMessage' in the class 'Client' and the method 'put' in the class 'Hashtable'. As a result, CCP is 12.

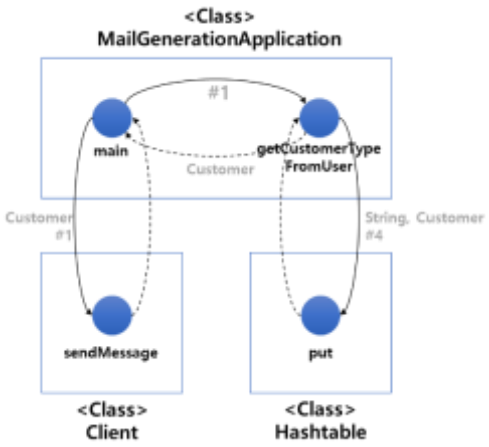
Table 3 A CCH & CCP Sample Code

Source Code

```
class MailGenerationApplication{
    private static Client client = new Client();
    public static void main(String[] args) {
        Customer customer = getCustomerTypeFromUser();
        MailGenerationApplication.client.sendMessage(customer);
    }
    private static Customer getCustomerTypeFromUser() {
        String customerType = "newbie";
        Hashtable customerTypeTable = new Hashtable();
        customerTypeTable.put("frequent", new Frequent());
        customerTypeTable.put("returning", new Returning());
        customerTypeTable.put("curious", new Curious());
        customerTypeTable.put("newbie", new Newbie());
        return null;
    }
}
```

Notation

Structure



<class> MailGenerationApplication

IIC

1

CCH

$$\sum_{i=0}^{IIC} MCP_i = \sum_{i=0}^{1} (CP - In_i + CP - Out_i) = (Data + Stamp) = 1 + 2 = 3$$

IOC

$$1 + 4 = 5$$

CCP

$$\sum_{i=0}^{IOC} MCP_i = \sum_{i=0}^{5} (CP - In_i + CP - Out_i)$$

$$= (Stamp + Data) + (Stamp + Data) + (Stamp + data) + (Stamp + Data)$$
$$= 3 + 3 + 3 + 3 = 12$$

3.2 Dynamic Metrics

We describe dynamic metrics as a measure of static metrics in a dynamic environment. Therefore, we omit detailed explanation.

4 Visualization System for identifying Reusable Code Modules

We develop our visualization system for automatically identifying reusable modules. Figure. 1 shows the whole structure of our system. The system consists of 5 steps: Input/Running, Analyzing, Storing, Reconstructing and Visualization. Each step is summarized as follows:

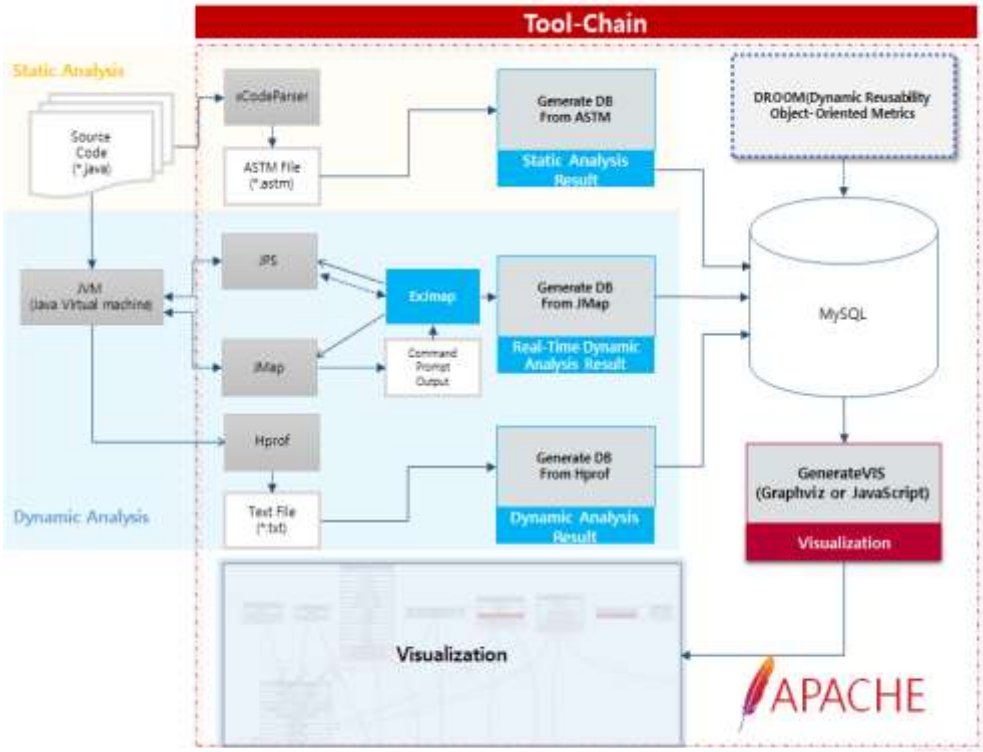


Figure. 1. Our Visualization System

- Step1 (Input/Running): This step perform Static Analysis (which analysis a source code without running a system), Real-Time Dynamic Analysis (which analysis a running system on the JVM) and Real-Time Dynamic Analysis (which analysis a running system and output the result at the end of a system).
- Step2 (Analyzing): This step use various tools. For a static analysis, we use xCodeParser (which is developed for static analysis). For a real-time dynamic analysis we use JPS (which collects executed process ID on JVM), JMap (which collect JVM Heap data), ExJmap (which is developed for real-time analysis). For a dynamic analysis, we use HPROF. Using these tools, we perform Static & Dynamic analysis.
- Step3 (Storing): This step analyses the parsing data to structure data and measures OORM. All data is stored in Database (MySQL).

- Step4 (Reconstructing): To visualize reusable modules, we reconstruct the data with a grammar of a visualization tool (Graphviz, GoJS, etc.).
- Step5 (Visualization): Using visualization tools, we visualize reusable modules. A user can see reusable modules in charts and images.

Through chaining the tools, we visualize the software architecture to identify reusability of each module. Thus, developers can automatically identify reusable module on the system that are provided with guidelines for reuse.

We show our best practice with the 'JUnit' [13]. Figure. 2 shows the result of automatic reusable module identification. In the architecture visualization, each node is a class, which displays the CCP, DOCP, CCH and DOCH inside the top row. Each edge is a call, which displays the frequency and CCP at this line. The classes are classified according to the color of the top row. Reusable modules are classified according to the color of the row of method and the edge. If each factor is lower than a criteria which is defined by customers, there are red. This color improve the visibility of users.

To verify OORM and the tool chain, we compare JUnit 3.4 and JUnit 4.0. Figure. 3 shows the reused method and not reused method. We check the reusability of these methods. Non reusable methods, which is identified by the tool chain, isn't reused. Method 'testAddTestSuite' and Method 'testCreateSuiteFromArray' are identified as non-reusable methods. In JUnit 4.0, these methods removed. Reusable methods, which is identified by the tool chain, is reused. Class 'ComparisonCompactorTest' is developed in JUnit 4.0. The methods in this class are identified as reusable modules.

5 Conclusion

Reuse is an important issue for improving software productivity, and reducing time and cost for a project. Previous works have been done on metrics for measuring the reusability of legacy code. However, they have some programs not to reflect the object oriented mechanism, and also to be biased on the static factors. To solve this problem, we extend the object oriented reusability metrics with the original metrics. Then, we develop the visualization system that automatically measures a reusability, which visualizes these factors within the inner code structure. This system provides a reuse guideline to developers. Therefore, it is possible to reduce the cost/time, and improve the quality on developing a new system based on the legacy system. In the near future, we will validate the metrics by applying reusability metrics to the well-known target software, and checking whether the identified modules are actually reused on upgraded versions or not.

Journal of Engineering Technology

Volume 7, Special Issue (Internet of Things),Oct. 2018, PP. 262-272

RESEARCH ARTICLE -ENGINEERNG TECHNOLOGY ASEE

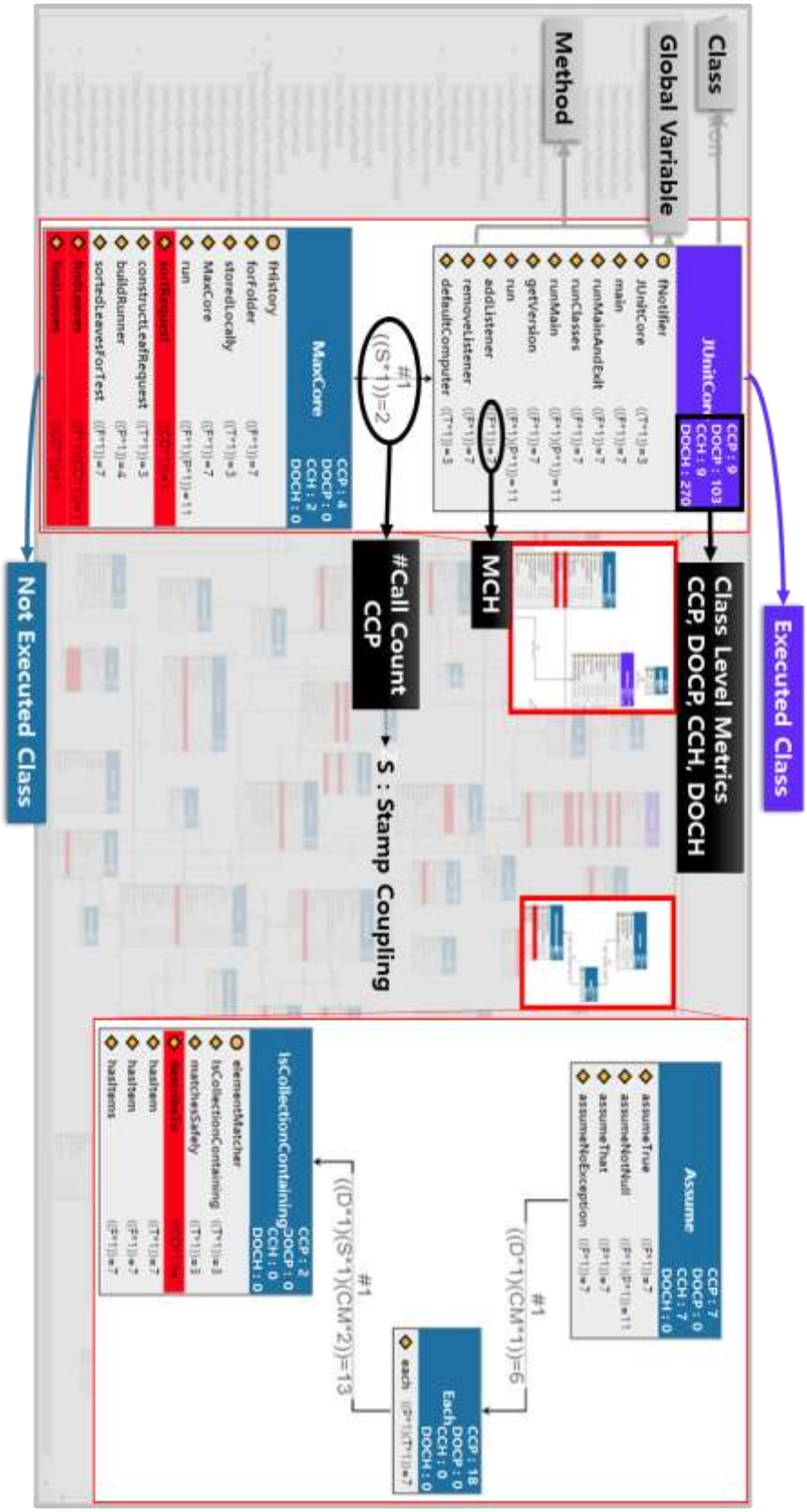


Figure. 2 Reusable Module Identification

Figure. 3 The comparison JUnit3.4 and JUnit 4.0

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A3B03035421), and this work was supported by the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (NRF-2015H1C1A1035548). Also, this work was supported by the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (NRF-2015H1C1A1035548)), and supported by a grant(18CTAP-C133299-02) from Technology Advancement Research Program funded by Ministry of Land, Infrastructure and Transport of Korean government.

References

[1]. Martin Shepperd, Darrel Ince (1993) Derivation and Validation of Software Metrics. International Series of Monographs on Computer Science. England.

[2]. J. M. Cowley (1995) Diffraction Physics 3rd Edition. Elsevier.

[3]. Thomas Ball, Stephen G. Eick (1996) Software visualization in the large. In IEEE Computer Society 29: 33-43.

[4]. Eun Young Byun, Hyun Seung Son, Byungkook Jeon, R. Young chul Kim (2018) Reusability Strategy Based on Dynamic Reusability Object Oriented Metrics. In Journal of Engineering Technology 6(1): 365-377.

[5]. Geon Hee Kang, R. Young Chul Kim, Geun Sang Yi, Young Soo Kim, Yong. B. Park, Hyun Seung Son (2015) A Practical Study on Code Static Analysis through Open Source based Tool Chains. In:KIISE Transactions on Computing Practices 21(2):148-153.

[6]. Marko Mijac, Zlatko Stapic (2015) Reusability Metrics of Software Components: Survey. In:Central European Conference on Information and Intelligent Systems : 221-230.

[7]. W Frakes, C. Terry (1996). Software reuse: metrics and models. In:ACM Computing Surveys 28(2): 415-435.

[8]. Shyam R. Chidamber, Chris F. Kemerer (1992) A metrics suite for object oriented design. In: International Financial Services Research Center, Sloan School of Management, Massachusetts Institute of Technology.

[9]. Ruchika Malhotra and Ravi Jangra (2017) Prediction & Assessment of Change Prone Classes Using Statistical & Machine Learning Techniques. In:Journal of Information Processing Systems 13(4): 778-804.

[10]. Kittakorn Sriwanna, Tossapon Boongoen, Natthakan Iam-On (2017) Graph clustering-based discretization of splitting and merging methods (GraphS and GraphM). In:Human-centric Computing and Information Sciences.

[11]. Ruchika Malhotra, Ankita Jain (2012) Fault Predicting Using Statistical and Machine Learning Methods for Improving Software Quality. In:Journal of Information Processing Systems 8(2):241-262.

[12]. Kevin Hoffman, Patrick Eugter (2008) Towards reusable components with aspects: an empirical study on modularity and obliviousness. In:International conference on Soft-ware engineering 91-100.

[13] JUnit, https://junit.org/

- Corresponding Author : Byungkook Jeon, R. Young Chul Kim