論文
2-3-15

Design of the Metrics Suite Π_{Java} for Java Program Complexity

(자바 프로그램의 복잡도 측정을 위한 척도 Π_{Java}의 설계)

김 은 미*
(Eun-Mi Kim)

ABSTRACT

In this paper we propose a suite of metrics Π_{Java}, for evaluating the complexity of Java programs based on a suite of metrics Π_{C++}, which we previously presented for C++ programs. So far, a lot of metrics for C++ are proposed for C++ programs. But since the specific properties of Java programs are not explicitly considered in those metrics, it is hard to apply them to Java programs. Thus we aim to develop a metric suite that is applicable to Java programs. At first, we decide if any properties are commonly possessed by both C++ programs and Java programs, or not. For example, the multiple inheritance of the class in C++ is not implemented in Java. On the other hand, the features such as package and interface are newly implemented in Java, and therefore we cannot discuss the complexity of Java programs without considering these new features. Then we define a new suite of metrics Π_{Java} for Java programs by deleting 3 metrics from Π_{C++}, and then incorporating 3 metrics which are newly defined or modified for Java programs to Π_{C++}. Finally, we analytically evaluate the new metric with regard to Weyuker's measurement principles and also compare it with conventional metrics for Java.

※ Keywords : object-oriented, Java program, software metrics, program complexity

요 약

본 논문에서는 C++ 프로그램을 측정하기 위하여 제안된 척도 Π_{C++}를 기반으로 Java 프로그램 복잡도를 측정하기 위한 척도 Π_{Java}을 제안한다. C++를 측정하기 위해 많은 척도들이 제안되어 왔으나 이들은 Java 프로그램만이 가지는 특성들을 고려하지 않았기 때문에 C++ 프로그램과 다른 성질을 가지고 있는 Java프로그램에 이들을 그대로 적용하는 것은 어렵다. 따라서, 본 논문에서는C++와 Java 언어의 공통점과 차이점을 분석한 결과와 본 연구팀이 C++ 프로그램의 복잡도를 측정하기 위해 제안했던 척도 Π_{C++}을 바탕으로 Java 프로그램의 복잡도를 측정하기 위한 새로운 척도 Π_{Java}을 제안한다. 마지막으로 제안한 척도를 Weyuker의 성질에 적용하여 분석하며, 기존의 척도들과도 비교한다.

※ 키워드 : 객체 지향, 자바프로그램, 소프트웨어 척도, 프로그램 복잡도

* 정회원 : 호원대학교 컴퓨터학부 조교수

논문접수 : 2001. 3. 28.
심사완료 : 2001. 4. 12.

※ 본 논문은 2000년 호원대학교 교내학술연구조성비에 의해 연구되었음.

1. Introduction

In 1980s we were brought a major breakthrough in software design with the introduction of object-oriented design method, technologies and language. Recently the object-oriented method has become very popular in software development because of its efficiency in software development processes and reusing software modules.

Several languages such as Eiffel, Modular-3, C++, Smalltalk and Java etc. that support object-oriented design and programming have been proposed. Among them C++, Java and Smalltalk are object-oriented language used widely for developing object-oriented software. C++ based on C language allows programs to be written in a hybrid style allowing non object-oriented code. Java and Smalltalk are developed as a pure object-oriented language. Java has most of the syntax of C++ but many of features of C++, such as multi-inheritance, have been removed or refined in order to promote true object-oriented programming. Java is designed to be simple, object-oriented and similar to C++ while removing the unnecessary complexities of C++[1]. Smalltalk treats every data as an object, including a number.

On the other hands, many software metrics have been introduced for analyzing productivity, reliability, maintainability, and complexity of software in order to reduce the costs and improve the quality of software products[2]. These metrics can also play a significant role when reverse engineering an existing software system.

In order to evaluate and support the object-oriented development, the object-oriented metrics have been introduced, since the traditional software metrics were not appropriate for applying the object-oriented development[3]. The object-oriented metrics are evaluated and newly proposed in much research [3,4,5,6,7].

Chidamber and Kemerer[3] have defined a suite of metrics for an object-oriented design. The metrics are based on measurement theory and confirmed by the insights of experienced object-oriented software developers. Li and Henry[4] have defined several object-oriented software metrics, evaluated the relationship among their metrics and applied them to analyze the maintenance effort in two commercial systems. Sharble and Cohen[7] have formally defined nine metrics, and applied them to compare two different development methods for an object-oriented software. However these metrics tend to measure the basis properties of object-oriented concept without regarding the unique properties of each object-oriented language, such as multi-inheritance in C++ or Eiffel.

We have proposed a suite of metrics Π_{C++} for C++, which tries to measure the unique properties attributed to C++ as well as the general object-oriented program[8]. The metric evaluates the following aspects of complexity: (1) Syntax complexity, (2) Inheritance complexity and (3) Interaction complexity. Then, we defined each key complexity as a function including five attributes, respectively.

In this paper, we propose a new suite of metrics Π_{Java} for evaluating the complexity of Java programs based on the suite of metrics C++ proposed in [8].

2. Preliminaries

2.1 Software metrics for JAVA

Java is a new object-oriented programming language close to C/C++. Using Java, developers can create dynamic and interactive programs that can run inside Web pages. Moreover, it can be used to create animations, games, and stand-alone applications[9].

Recently, some evaluation metrics for Java programs have been proposed together with that tools developed for them[10,11]. J-Metric[10]

proposed by Cain and Vasa measures the complexity of Java as the units of project, package, class, method and variable. For the project metric, it measures the number of classes, methods, variables, lines of code, statement count, the number of Inner classes and the maximum depth of inner classes. For package metric, it measures the number of classes, methods, variables and summarized list of the main metrics from classes, methods inside this package. The number of methods and variables, lack of cohesion of method, collaborators for a class, lines of code, statement count, inner classes and inheritance related metrics are measured for the class metrics. Also, it measures cyclomatic complexity, statement count and lines of code, instant variables and the number of local variable as the method metric. Finally, the number and times used of instance variable are measured as the variable metrics.

Next, Banda Java Source Metric[11] evaluates Java program as the unit of the class, method, application, person and scheduling. For the class and method, it evaluates the complexity for size, the complexity of inheritance and the internal complexity. For the application and person, it evaluates the complexity for size.

Finally The metrics proposed by Chidamber and Kemerer are often used to evaluate the complexity of Java program[9]. Chidamber and Kemerer defined the six metrics : Weighted method per class, Depth of inheritance tree, Number of children, Coupling between objects, Response for a class and Lack of cohesion in methods.

However the attributes measured in these tools are not different from the conventional metrics of object-oriented language and cannot evaluate the characteristic parties of Java exactly.

2.2 A Suite of Metrics (C++

2.2.1 Outline

We proposed a suite of metric Π_{C++} for computing the complexity of an object-oriented program[8]. The metric evaluates the following aspects of the object-oriented paradigm: (1) Syntax complexity, (2) Inheritance complexity and (3) Interaction complexity. Then, we expanded each key complexity to five attributes in detail.(With respect to the detail of attributes, please refer to [8].)

$COMP(P) = f(SX(P), IH(P), IT(P))$
where
Syntax complexity of program P :
$SX(P) = f1(IMC, NOM, NOCL, LCOM, UOC)$
Inheritance complexity of program
$P : IH(P) = f2(DIT, NODC, NODA, DOR, NOD)$
Interaction complexity of program P : $IT(P)$
$= f3(CBI, RFC, UCL, VOD, MPC)$

$COMP(P)$ evaluates the complexity of a given program P, as a function f of three parameters: $SX(P)$, $IH(P)$ and $IT(P)$. $SX(P)$, $IH(P)$ and $IT(P)$ evaluate the complexity of each dimension by computing values for each attribute. Here, we use the definition of software complexity defined by Zuse[2]. That is, we define software complexity as the effort or time which is consumed to maintain the program.

2.2.2 Attributes for syntax complexity

In order to evaluate the syntax aspect of a program, we introduced the following five attributes which corresponded to the size of a program and the coding efforts. Here, *IMC*, *NOCL* and *UOC* were newly proposed in [8].

① $IMC(M_i)$: Degree of internal method complexity for a method M_i.

② *NOM*(C_i)[3]: Number of methods in a class Ci.
③ *NOCL*(*P*): Number of classes in a program *P*.
④ *LCOM*(C_i)[3]: Degree of lackness of cohesion in a class *Ci*.
⑤ *UOC*(*P*): Ratio of used classes to defined classes in a program *P*(0<*UOC*(*P*)1).

2.2.3 Attributes for inheritance complexity

In order to evaluate the inheritance aspect of a program, we introduced the following five attributes which corresponded to the degree of reuse by inheritance. Here, *NODA*, *DOR* and *NOD* were newly proposed in [8].

① *DIT*(C_i)[3]: Depth of inheritance trees for a class Ci.
② *NODC*(C_i)[3]: Number of children of a class Ci.
③ *NODA*(C_i): Number of all inheriting ancestors of a class *Ci*.
④ *DOR*(C_i): Degree of reuse by inheritance (0*DOR*(Ci)<1) for each *Ci*.
⑤ *NOD*(*P*): Number of disjoint inheritance trees in a program *P*.

2.2.4 Attributes for interaction complexity

In order to evaluate the interaction aspect of a program, we introduced five attributes which corresponded to the degree of coupling. Specifically, *CBI* reflects the 'is kind of' relationship, and *UCL* reflects the 'is part of' relationship. Here, *CBI* and *UCL* were newly proposed in [8].

① *CBI*(C_i): Degree of coupling of inheritance in a class *Ci*.
② *RFC*(C_i)[3]: Degree of response in a class *Ci*.
③ *UCL*(C_i): Number of classes used in a class Ci except for ancestors and children.
④ *VOD*(C_i)[7]: Number of violation of the law of Demeter in a class *Ci*.
⑤ *MPC*(C_i)[4]: Number of send statements in a class Ci.

3. A New Suite of Metric (Java

3.1 Comparison between C++ and Java

Java has most of the syntax of C++ but many of features of C++, such as multi-inheritance, have been removed or refined in order to promote object-oriented programming. Therefore, there exist some similarities and differences between C++ and Java.

We analyze the two languages from the static viewpoint because the metric we try to propose is considered the static viewpoint of the language. <Table 1> shows the similarities and differences between Java and C++. Here, ○ means each item exists in the language, × means the item doesn't exist in the language.

While all function and method definitions in Java are contained within the class definition, in C++, functions can be defined as stand-alone or global or member of a class. Next, Java can call functions written in another language, commonly referred to as native method. C++ can also call functions written in C, but it is not general. Java supports classes, but doesn't support structures or unions. Java doesn't also manipulate operator overloading, pointers, typedef, define and goto statement.

Java also allows single inheritance only. Thus, each class can have only one superclass. When a subclass is created, we have to define only the differences between that class and its superclass. By default, a new class with no explicit inheritance inherits from the Object class, the top class in Java. When an object is created, a slot for each non-static variable defined in the object class and in all its superclasses is reserved[12]. But, in C++ multiple inheritance enables a programmer to derive a class

from multiple parent classes.

Moreover, Interfaces provide templates of behavior that other classes are expected to implement. A class, in Java, can implement one or more interfaces. Package in Java is similar to #include in C and the classes of Java must belong to packages.

3.2 Outline of new metrics suite Π_{Java}

The new metric suite (Java includes or modifies the attributes which is extracted from the previous metric suite Π_{C++} for C++. Also, the new metric suite (Java includes the additional ones that measure the unique properties for Java. Thus, the new metric suite (Java is defined as the following:

M_{common} = {*NOM, LCOM, DIT, NODC, DOR, RFC, UCL, VOD, MPC*}

M_{C++}={*IMC, NODA, NOD, CBI*}

M_{Modify}={*NOCL, UOC*}

M_{Java}={*NOP*}

NOP = the number of package

A suite of metric $\Pi_{C++} = M_{common} \cup M_{Modify} \cup M_{C++}$

A suite of metric Π_{Java}

$= M_{common} \cup M\text{Modify} \cup M_{Java}$

Here, M_{common} represents a set of attributes that are commonly shared by C++ and Java. M_{C++} represents a set of that are attributes included in C++ metric but deleted in Java metric. M_{Modify} represents a set of that are attributes included in C++ metric but modified in Java metric. M_{Java} represents a set of attributes that are added newly in Java metric.

From <Table 1>, all functions in Java exist only the member of the class. So, to measure the complexity of a method is meaningless, we exclude *IMC* from the new metric. Since Java doesn't support multiple inheritance, the attributes *NODA*,

<Table 1> Comparison between Java and C++

Items		Java	C++
Function	Main	m	m
	Stand-alone	×	m
	Global	×	m
	Method (member of a class)	m	m
Calling functions written in another language		m	m
Class		m	m
Struct or Union		×	m
Inheritance		Single	Multi
Operator overloading		×	m
Pointer		×	m
Interface		m	×
Access control (private, public, protected)		m(the meaning is different)	m
Typedef or define		×	m
Goto statement		×	m
Package		m	m

m: The item exists, × : The item doesn't exist

NOD and *CBI* are deleted from the new metric.

Among the characteristics of Java, the interface and package are important factors which influence the complexity of Java (we explain the reason in the section 4). So, we adopt the new attribute *NOP* and modify two attributes *NOCL* and *UOC* to evaluate Java.

4. Heuristics Procedures

We explain the heuristics for new attributes in MModify and MJava proposed in the previous section.

4.1 Procedure *NOCL*

The complexity of program is related to the number of classes, and the number of classes is related to the degree of reuse of class. The lager the number of classes is, the greater the efforts of programmer and the less the degree of reuse of class is.

Generally, all the classes in Java are classified into three types : class, interface and exceptions. A class consists of attributes and behavior. Attributes are individual characteristics used to differentiate one object from another and usually describe the object's state. An interface is a program unit, such as class. It is similar to a class, but with only declarations of its methods. Its behavior is similar to that of abstract class[1]. In Java program, differences between interfaces and abstract classes are summarized as follows: interface provides a form of multiple inheritance, but a class extends only one class[13]. Therefore, the complexity of program is related to the number of interfaces, and the number of interfaces is related to the degree of inheritance. Finally, exception is an event happening during execution of a program that disrupts the normal flow of control.

However these classes have different contribution to the program respectively. For example, It is not sufficient that the interface is only defined. The declaration and use of interface are different problem. Generally, some of the defined interfaces can be used in the program P. Therefore NOCL is computed by the following weighting formula.

$$NOCL(PA) = \text{the number of class} \times w_{c1} + \text{the number of interface} \times w_{c2} + \text{the number of exception} \times w_{c3}$$

Here, NOCL is computed for a package PA, and wc_1, wc_2 and wc_3 are the weighting constants.

4.2 Procedure UOC

Sometimes, It is not sufficient that the class or interface is only defined. The declaration and use of them are different problem. Generally, some of the defined class can be used in the program *P*. In this case, we consider that the usability of the class decreases and the complexity of use of the class increases.

However an interface defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior. Therefore interface is useful to implement the large program which is composed of several related modules. Then the use of interface can decrease the complexity of the program. Thus *UOC* is also computed by the following weighting formula.

$$UOC(P) = \frac{\text{The number of used class} \times w_{u1} + \text{The number of used interface} \times w_{u2}}{\text{The number of defined class} + \text{The number of defined interface}}$$

Here, UOC is computed for a program P, and w_{u1} and w_{u2} are the weighting constants.

4.3 Procedure *NOP*

Software maintenance requires understanding existing code. Java programs are organized as sets of packages. To understand the entire program, we should understand the packages which are included in a program. Therefore, the complexity of a program is influenced by the packages in a program. The larger the number of packages, the more complex the program is. Packages use its fully qualified name, or import all or part of the package. NOP is the number of package and is computed for a program P.

NOP(P) = the number of *package* declaration $\times$ w_{p1}
+ the number of *import* declaration $\times$ w_{p2}
+ the number of packages used in the declaration of class or interface type $\times$ w_{p3}

Here, w_{p1}, w_{p2} and w_{p3} are the weighting constants.

5. Discussion

5.1 Analysis using Weyuker's Properties

In this section, we adopt the Weyuker's properties[14] to evaluate the new metric. Weyuker's properties suffer the following criticisms from some people[3,15] : First, Zuse pointed out that Weyuker's properties are not consistent with the principles of scaling. Next, Cherniavisky and Smith suggested that we should be careful with Weyuker's properties, since the properties may give only necessary conditions for good complexity metric. However, since these properties are widely known and accepted, we also adopt them to evaluate the new metric.

<Table 2> shows the evaluation results of our proposed attributes with regard to Weyuker's properties. In the following analysis, we will discuss evaluation result only for the attribute, NOP among them that we defined. In addition, we will exclude discussions on Property 2 and Property 8 in this paper. Since the universe of discourse deals with at most a finite set of applications, each of which has a finite number of classes and methods, Property 2 will be met by any metric measured at the class level[3]. Next, since none of the metrics proposed in this paper depend on the names of the class or methods and instance variables, they also satisfy Property 8.

Property 1 reflects an intuition that any measure should not rank all programs as equally complex. Clearly, Property 1 is satisfied by all of our proposed attributes. Any measure, which assigns a unique numerical name to each program and treats this name as the program's complexity, would fail to satisfy this property[14]. Clearly, Property 3 is satisfied by all of our proposed attributes.

Properties 1, 2 and 3 are exactly the real properties to be satisfied by measures and do not directly reflect the fact that we are dealing with programs which have syntax and semantics[14]. All the proposed attributes satisfy these properties. Thus, we can conclude that the proposed attributes have essential properties as software metrics.

Property 4 intuitively implies that even if two programs compute the same function, the complexity of these programs can be different and be determined based on the details of their implementations. From a pragmatic point of view, Property 1 and Property 4 are essentially equivalent[2]. Since the NOP satisfy Property 1, it also satisfies Property 4.

Consider the attribute NOP. Let P and Q be two programs with n and m packages respectively. That is, $|P|=n$ and $|Q|=m$. $|P;Q|=n+m-p$, $|P| \leq |P;Q|$ and $|Q| \leq |P;Q|$. Here, p is the number of packages that P and Q have in common and the range is $0 \leq p \leq n$(if

n is smaller than m). Therefore, Property 5 is also satisfied by the NOP.

Assume that |P|=|Q|=n, |R|=m for the attribute NOP. Assume also that a package is commonly included in programs P and R, and the sets of packages in programs Q and R are disjoint. Then |P;R|=n+m-1 and |Q;R|=n+m. Thus, |P|=|Q| and |P;R| ≤|Q;R| hold and so Property 6 is satisfied

Property 7 asserts that program complexity should reflect the order of statements in the program, and hence the potential interaction among statements. Since the order of statements within class or method is not related to actual execution or use of class or method, Property 7 is not satisfied by any attribute proposed in this paper.

Property 9 reflects the fact that interaction may exist between concatenated subprograms. In such case, the attribute NOP satisfy Property 9.

As the result of evaluation, all attributes proposed by our metric satisfy Properties 1, 2, 3, 4, 5, 6 and 8. Cherniavisky and Smith described that Property 7 is not appropriate for object-oriented metrics, and in fact the proposed attributes do not satisfy Property 7. Moreover, some of the attributes do not satisfy the Property 9. These property allows for the possibility that, as a program grows from its component program, additional complexity is introduced due to the potential interactions among these component parts[14].

Chidamber suggested that failure in satisfying Property 9 implies that a complexity metric could increase when classes are divided into many more subclasses. Also, experienced object-oriented designers found that memory management and run-time detection of errors are both more difficult when there are a large number of classes to deal with[3]. Therefore, satisfying Property 9 may not be an essential feature for an object-oriented software design complexity metric[3]. We agree with him. In order to confirm the opinion, we will examine Property 9 through experimental evaluations using a data regarding the development process and software products collected from practical software development projects.

<Table 2> Evaluation result of Weyuker's properties

Property / Attribute	1	2	3	4	5	6	7	8	9
NOP	m	m	m	m	m	m	×	m	×
NOM	m	m	m	m	m	m	×	m	×
NOCL	m	m	m	m	m	m	×	m	×
LCOM	m	m	m	m	m	m	×	m	×
UOC	m	m	m	m	×	m	×	m	m
DIT	m	m	m	m	×	m	×	m	×
NODC	m	m	m	m	m	m	×	m	×
DOR	m	m	m	m	×	m	×	m	×
RFC	m	m	m	m	m	m	×	m	×
UCL	m	m	m	m	m	m	×	m	×
VOD	m	m	m	m	×	m	×	m	×
MPC	m	m	m	m	×	m	×	m	×

<Table 3> Comparison of Java metrics

	Program	Package	Class	Interface	Method	Variable
J-Metric	m	m	m		m	m
BJSM	m		m			m
C&K Metric	m		m		m	m
New Metric	m	m	m	m	m	m

BJSM : Banda Java Source Metric, C&K Metric : Metric proposed by Chidamber and kemerer

5.2 Comparison

In this chapter, we compare the proposed metric with the conventional metrics for Java.

From the <Table 3>, we can see that. J-Metric does not consider the metric with regard to the interface. BJSM does not consider the metric for the package, interface and method for Java. C&K metric also does not consider the metric for package and interface of Java.

Therefore we can see the proposed metric evaluates the unique characteristics of Java well.

6. Conclusion

In this paper, we propose a new suite of metrics Π_{Java} for evaluating the complexity of Java programs based on the metric suite Π_{C++} which we proposed. Also we analytically evaluate the new metric with regard to Weyuker's measurement principles and also compare it with conventional metrics for Java.

Currently, as the future research work, we are planning to develop the tool for calculating the values of attribute automatically. Also, we decide the weight for each metric and evaluate the usefulness of the proposed metric Π_{Java} using the tool.

※ REFERENCES

[1] G. James, The Java language environment : http: //java.sun.com/doc /language_ environ-ment/.

[2] H. Zuse, Software complexity-measures and methods, 1991.

[3] S. R. Chidamber and C. F. Kemerer, "A metric suite for object-oriented design," IEEE Trans. On Softw. Eng, 20, 6, pp. 476-493, 1994.

[4] W. Li and S. Henry, "Object-oriented metric that predict maintainability," The Journal of Systems and Software, 23, pp.111-122, 1993.

[5] D. R. Moreau and W. D. Dominick, "Object-oriented graphical information systems : Research plan and evaluation metric," The Journal of Systems and Software, 10, 1, pp.23-28, 1989.

[6] S. L. Pfleeger and J. D. Palmer, "Software estimation for object-oriented program", Proc. of FPUGFC90, pp.181-190, 1990.

[7] R. C. Sharble and S. S. Cohen, "The object-oriented brewery: A comparison of two object-oriented development methods", ACM SIGSOFT Software Engineering Notes, 18, 2, pp.60-73, 1993.

[8] E. M. Kim, S. Kusumoto and K. Kikuno, "A new metric for C++ program complexity and its evaluation in academic environment," IEICE trans. J79-D-1, 10, pp.729-737, 1996.

[9] T. Systa and P. Yu, " Analyzing java software by combing metrics and program visualization," Proc. of the Conference on Software Maintenance and Reengineering, pp. 1-10,1998.

[10] A. Cain and R. Vasa , Java Metric Analyser, http://jmetric.it.swin.edu.au/products/ jmetric/.

[11] W. B. Brian : BANDA Java Packages, http://www.sladen.com/Java/docs/ banda/metric/.

[12] A. S. boujarwah, K. Saleh and J. Al-Dallal, "Dynamic data flow analysis for Java programs," The Journal of Information and Software Technology, 42, pp. 765-775, 2000.

[13] G. James, Java language specification, http:// java.sun.com/docs/books/jls/html/.

[14] E. J. Weyuker, "Evaluating software complexity measures," IEEE Trans. on Softw. Eng., 14, 9, pp.1357-1365, 1988.

[15] J. C. Cherniavisky and C. H. Smith, "On Weyuker's axioms for software complexity measures," IEEE Trans. on Softw. Eng., 17, pp636-638, 1991.

김은미

1991년 전북대학교 전산통계학과 졸업

1993년 전북대학교 대학원 전산통계학과 졸업(이학석사)

1997년 일본 오오사카대학 기초공학연구과 정보공학전공 졸업(공학박사)

현재 : 호원대학교 컴퓨터학부 조교수

관심분야 : 객체지향 시스템, 소프트웨어 척도, 소프트웨어 검증방법