한국산학기술학회논문지
Vol. 10, No. 12, pp. 3715-3724, 2009

Component Metrics to Measure Component Quality

Chul-Jin Kim[1] and Eun-Sook Cho[2*]
[1]**Dept. of Computer System, Inha Technical College**
[2]**Dept. of Software, Seoil University**

컴포넌트 품질 측정을 위한 컴포넌트 메트릭

김철진[1], 조은숙[2*]
[1]인하공업전문대학 컴퓨터시스템과
[2]서일대학 소프트웨어과

Abstract Recently, component-based software development is getting accepted in industry as a new effective software development paradigm. Since the introduction of component-based software engineering (CBSE) in later 90's, the CBSD research has focused largely on component modeling, methodology, architecture and component platform. However, as the number of components available on the market increases, it becomes more important to devise metrics to quantify the various characteristics of components. In this Paper, we propose metrics for measuring the complexity, customizability, and reusability of software components. Complexity metric can be used to evaluate the complexity of components. Customizability is used to measure how efficiently and widely the components can be customized for organization specific requirement. Reusability can be used to measure the degree of features that are reused in building applications. We expect that these metrics can be effectively used to quantify the characteristics of components.

요 약 최근 들어 산업계에서 컴포넌트 기반의 소프트웨어 개발이 새로운 효율적 소프트웨어 개발 패러다임으로 받아들여지고 있다. 1990년대 후반 컴포넌트 기반 소프트웨어 공학이 소개되면서 컴포넌트기반 소프트웨어 개발(CBSD) 관련 연구는 컴포넌트 모델링, 개발 방법론, 아키텍처, 그리고 컴포넌트 플랫폼 등에 주로 집중되어왔다. 그러나 시장에서 가용한 컴포넌트들의 수가 증가함에 따라, 컴포넌트들의 다양한 특성들을 정량화하기 위한 메트릭에 대한 개발이 점차 중요해지기 시작했다. 본 논문에서 우리는 소프트웨어 컴포넌트의 복잡도, 특화성, 재사용성을 측정할 수 있는 메트릭들을 제안한다. 복잡도 메트릭은 컴포넌트의 복잡성을 평가하는데 사용가능하고, 특화성은 해당 컴포넌트가 조직의 특화된 요구사항에 맞도록 얼마나 효율적이면서 폭넓게 커스터마이즈될 수 있는지를 측정하는데 사용된다. 재사용성은 애플리케이션을 구축할 때 해당 컴포넌트의 재사용되는 정도를 측정하는 용도로 사용된다. 제안하는 이러한 메트릭들은 컴포넌트가 갖는 특징들을 정량화하는데 보다 효율적으로 사용될 수 있으리라 기대한다.

Key Words : Component, Component-Based Development, Customizability, Reusability, Component Metric

1. Introduction

Object technologies have been often heralded as the silver bullet for solving software reuse problems since early 1980. However, it's been known that objects are too small-grained units, especially for enterprise application development projects. Component technology has been introduced with new approach to address reusability problem in software development. Various component platforms such as COM+, EJB, and CCM, component modeling techniques, component development tools, and component development processes were introduced [1,3,5].

Component-oriented software development requires a

*Corresponding Author : Eun-Sook Cho(escho@seoil.ac.kr)
Received November 08, 2009 Revised December 04, 2009 Accepted December 16, 2009

considerably different approach from object-oriented (OO) methods[4]. While OO methods develop systems by defining functional and object models, component-based development (CBD) methods utilizes commonality and variability (C&V) analysis, components, component's interfaces, and relationships among components [1]. Therefore, various metrics developed for OO programming cannot be equally applied to CBD process. Hence, in this paper, we propose component metrics that can be efficiently applied in CBD process.

The paper is organized as follows. We first discuss relevant OO metrics. These metrics focus on object structure that reflects the complexity of each individual entity, such as methods and classes, and on external complexity that measures the interactions among entities, such as coupling and inheritance. Then, we show the limitations of existing OO metrics in applying to CBD. In chapter 3, we propose three metrics to measure component's quality; complexity, customizability, and reusability. We define each metric and suggest the applicability of each metric in CBD. Chapter 4 presents a case study conducted with the proposed metrics. Also, we compare proposed metrics to existing metrics.

2. Related Works

2.1 Metrics for Object-Oriented System

Many different metrics have been proposed for object-oriented systems.

The object oriented metrics measure principle structures that, if improperly designed, negatively affect the design and code quality attributes[2,6,12]. Existing object oriented metrics are primarily applied to the concept of classes, coupling, and inheritance[11].

2.2 Weighted Methods per Class (WMC)

The WMC is a count of the methods implemented within a class or the sum of complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement since not all methods are assessable within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effortis required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children; children inherit all of the methods defined in the parent class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse [7-10].

2.3 Limitations of Existing OO Metrics

In this section, we discuss difficulties of applying existing object-oriented metrics into component development and CBSD. It is not adequate in measuring component's qualification with object-oriented metrics themselves discussed in previous section. The reason is that as following:

1. Measurement unit is different. OO metrics only focus on objects or classes. Component consists of one or more classes as well as one or more interfaces. Existing object-oriented metrics do not consider component itself or component's interfaces on measuring complexity, cohesion, or coupling, and so on. Therefore, it is required new metrics that measure complexity of component itself.

2. Measurement factor is insufficient. Because object-oriented applications are developed with only classes, almost OO metrics measure the complexity or reusability by considering classes, methods, and depth of class hierarchy. However, considering only these factors is not adequate to measure the complexity or reusability of component because components have more much information such as interface, interface methods, and so on. While existing OO metrics do not consider customizability of classes or objects, customizability of component is very important in CBD because component's customizability effects on reusability of components in CBD.

3. Definition of Component Metrics

We will propose some metrics to measure complexity, customizability, and reusability in this chapter. We define metrics to measure the quality of designed components as well as we propose metrics for measuring the quality of implemented components[13]. Therefore, proposed metrics

are classified into design metrics and implementation metrics.

3.1 Measuring the Complexity

To measure the complexity, the cyclomatic complexity is used in traditional program. Cyclomatic complexity (McCabe) is used to evaluate the complexity of an algorithm in a method. It is a count of the number of test cases that are needed to test the method comprehensively. The formula for calculating the cyclomatic complexity is the number of edges minus the number of nodes plus 2. A method with a low cyclomatic complexity is generally better. Cyclomatic complexity cannot be used to measure the complexity of a component because of inheritance in a component, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the component. Therefore, we propose new complexity metric to measure complexity of a component by combining cyclomatic complexity: Component Complexity Metric (CCM). We classify CCM into four kinds of complexity metrics: component plain complexity (CPC), component static complexity (CSC), component dynamic complexity (CDC), and component cyclomatic complexity (CCC). While component cyclomatic complexity of these component complexity metrics is used in component implementation phase, other complexity metrics can be applied in component design phase.

3.1.1 CPC

The first approach used in order to measure the complexity of each component is CPC. CPC is a metric that measures the complexity of component itself by calculating the sum of classes, abstract classes, and interfaces, and the complexity of classes and methods. CPC is expressed by the following formula:

[Def.1]

$$CPC(C) = CmpC + \sum_{i=1}^{m} CC_i + \sum_{j=1}^{n} MC_j$$

where:

CmpC: is calculated by counting classes, abstract classes, and interfaces,

$\sum_{i=1}^{m} CC_i$: the complexity of each class, and

$\sum_{j=1}^{n} MC_j$: the complexity of each method.

The CmpC is calculated by counting classes, abstract classes, interfaces, and methods. The definition of CmpC is given by:

[Def.2]

$$CmpC = \sum_{i=1}^{m} (Count(C_i) \times W(C_i)) + \sum_{j=1}^{n} Count(I_j) + \sum_{k=1}^{o} (Count(M_k) \times W(M_k))$$

where:

Count(C): The count of the class of contained in a component,

W(C): weight value of each class,

I: the interface of provided/used by a component,

Count(M): The count of the methods of classes contained in a component, and

W(M): weight value of each method.

Classes contained in a component are divided into internal classes and external classes. External classes are imported classes from other reused library or packages. Internal classes are identified classes during component analysis and design in a domain. We give weight value to internal classes because external classes are implemented classes. Also, methods of internal classes are given weight value because methods of external classes are only invoked.

The complexity of each class (CC) contained in a component is calculated by counting single attributes of each class as Single Attribute (SA) and complex attribute, (i.e. attribute which type is a class), as Complex Attribute (CA). Then, we define CC as following formula:

[Def.3]

$$CC = \sum_{i=1}^{m} (Count(SA_i) + \sum_{j=1}^{n} (Count(CA_j) \times W(CA_j))$$

where:

Count(SA): The count of single attribute,

Count(CA): The count of complex attribute,

W(CA): Weight value of each complex attribute.

The complexity of each method of classes is calculated by counting parameters of each method. Simple argument is counted as SP, while complex argument, such as objects, is counted as CP. Also, complex arguments are given with weighted value because complex arguments contain another arguments in it. MC is given by following formula:

[Def.4]

$$MC = \sum_{i=1}^{m} Count(SP_i) + \sum_{j=1}^{n} (Count(CP_j) \times W(CP_j))$$

where:

Count(SP): The count of single argument,

Count(CP): The count of complex argument, and

W(CP): Weight value of each parameter.

3.1.2 Component Static Complexity

The second approach used in order to measure the complexity of each component is CSC. CPC only focuses on the number of classes, interfaces, methods, and parameters declared in a component, while CSC focuses on how complex the component's the internal structure. CSC is a metric that measures the complexity of internal structure in a component with a static view. Therefore, the static complexity of each component is calculated by counting relationships among classes contained in a component. We define the CSC as following formula:

[Def. 5]

$$CSC = \sum_{i=1}^{m} (Count(R_i) \times W(R_i))$$

where:

Count(R): The count of each relationship between classes, and

W(R): Weight value of each relationship.

There are four relationships between classes as UML specification.[4] According to accessibility between classes, the size of weight vale for the relationships is defined. We give the weight value as following priority:

Dependency<Aggregation<Generalization<Aggregation <Composition.

On counting relationships, if there are n-ary relationships among classes, n-ary relationship should be converted into binary relationship.

3.1.3 Component Dynamic Complexity

The third approach used in order to measure the complexity of a component is CDC.

CSC only focuses on how complex the component's the internal structure, while CDC focuses on how many message passing is occurred in a component. CDC is a metric that measures the complexity of internal message passing in a component with a dynamic view. Therefore, the dynamic complexity of each component is calculated by counting messages passed between classes contained in a component. We define the CDC as following formula:

[Def. 6]

$$CDC = \sum_{i=1}^{m} DC(IM_i)$$

where:

$\sum_{i=1}^{m} DC(IM)$: the complexity of each interface method.

[Def. 7]

$$DC(IM) = \sum_{i=1}^{n} (Count(Msg_i) \times Freq(Msg_i) + MC(Msg_i))$$

where:

Msg: the message passed between classes,

Freq(Msg): the frequency of messages passed between classes, and

MC(Msg): the complexity of each message, equal to the MC defined in [Def. 4].

3.1.4 Component Cyclomatic Complexity

The fourth approach used in order to measure the complexity of a component is CCC. While previous three metrics (i.e. COFP, CSC, CDC) are used in a component design time, CCC is used after the component implementation is finished. Therefore, other three metrics are calculated by using class diagram, interaction diagram, and component diagram, while CCC is computed by using developed source code. The difference between CPC and CCC is that the complexity of interface method declared in the interface of a component is based on cyclomatic complexity metric used in traditional program. CCC is defined as following formula:

[Def.8]

$$CCC = CmpC + \sum_{i=1}^{m} CC_i + \sum_{j=1}^{n} MC_j + \sum_{k=1}^{o} CCM_k$$

where:

CmpC: the sum of classes, interfaces, and interface methods defined in [Def. 2],

$\sum_{i=1}^{m} CC_i$: the sum of complexity of each class contained in a component, and

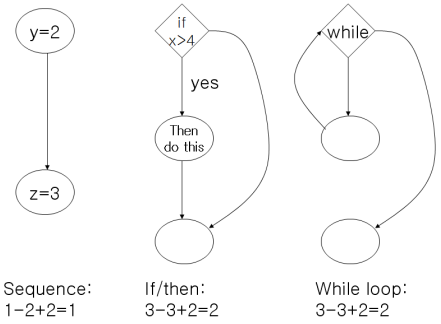
$\sum_{j=1}^{n} MC_j$: the sum of complexity of each interface

method.

The complexity of each class and of each interface method is equal to the [Def. 3] and [Def. 4]. However, the cyclomatic complexity of each method implemented in a class may be computed because CCC is calculated by using implemented component source code. We define the cyclomatic component method as following formula:

[Def. 9] $= \sum_{k=1}^{o} CCM_k = edges - nodes + 2$

[Def.9] is referred to [9]. The formula for calculating the cyclomatic complexity is the number of edges minus the number of nodes plus 2.



[Fig. 1] Example of Cyclomatic Complexity

For a sequence where there is only one path, no choices or option, only one test case is needed. An IF loop however, has two choices, if the condition is true, one path is tested; if the condition is false, an alternative path is tested. Figure 1 shows examples of calculations for the cyclomatic complexity for four basic programming structures.

3.2 Measuring the Customizability

The one of component's characteristics is component customization. If a component does not provide customizable interfaces, reusability of a component becomes low because application developers want to customize reusing components according to theirs purpose. Therefore, customizability of component should be considered in a component development process. In this chapter, we present customizability metric may be used in a component design phase or after the component development.

In order to measure customizability, we use component's variability methods as following formula:

[Def. 10]

$$CV = \frac{\sum_{i=1}^{m} Count(CVM_i)}{\sum_{j=1}^{n} Count(CIM_j)}$$

where:

CV: Component variability to measure customizability,

Count(CVM): the count of method for customization

Count(CIM): the count of method declared in each interface

According to [Def. 8], CVM is redefined as following formula:

[Def.11]

$$CVM = \sum_{i=1}^{m} (Count(CVMa_i)) + \sum_{j=1}^{n} Count(CVMm_j) + \sum_{k=1}^{o} Count(CVMw_k))$$

where:

Count(CVMa): the count of the method for attribute customization,

Count(CVMm): the count of the method for behavior customization,

Count(CVMw): the count of the method for workflow customization,

W(CVMm): Weight value for behavior customization method, and

W(CVMw): Weight value for workflow customization method.

As given in [Def. 11], we assign weight value into behavior customization methods and workflow customization methods. The reason is that those methods are more complex than attribute customization methods. Furthermore workflow customization methods are more complex than behavior customization methods because they contain several business methods in a workflow customization. Then, the priorities are given as following order: attribute customization method<behavior customization<workflow customization method.

3.3 Measuring the Reusability

We propose two approaches to measure the reusability of component in this paper. The one is a metric that measures how a component has reusability, while the other is a metric that measures how a component is reused in a particular application.

The first approach is component itself reusability (CR). CR metric may be used at design phase in a component development process. CR is calculated by dividing sum of interface methods providing commonality functions in a domain into the sum of total interface methods. We define the CR as following formula:

[Def. 12]

$$CR = \frac{\sum_{i=1}^{n}(Count(CCM_i))}{\sum_{j=1}^{m} Count(CIM_j)}$$

where:

Count(CCM): The count of each interface method for providing common functions among several applications in a domain, and

Count(CIM): The count of methods declared in interfaces provided by a component.

The second approach is a metric to measure particular component's reuse level per application in a CBSD. We call that Component Reuse Level (CRL). CRL is divided into CRLLOCs and CRLFunc. While CRLLOCs is measured by using Lines of Code (LOC), CRLFunc is measured by dividing functionality that a component supports into required functionality in an application.

The CRLLOCs, expressed as a percentage, for a particular application is given by:

[Def. 13]

$$CRLLOCs(C) = \frac{Reuse(C)}{Size(C)} \times 100\%$$

where:

Reuse(C): The lines of code reused component in an application,

Size(C): The total lines of code delivered in the application.

The CRLFunc is expressed with:

[Def. 14] CRLFunc(C)= Sum of supported functionality in a component/Sum of required functionality in an application

According to the [Def.14], the more many functions are supported in a component, the more much the reusability of a component in an application. If we apply this metric to a component used different applications for the same domain, we get the reusability of a component in a domain.

4. Case Study

This is the conclusions for our paper.4. Case Study and Assessment

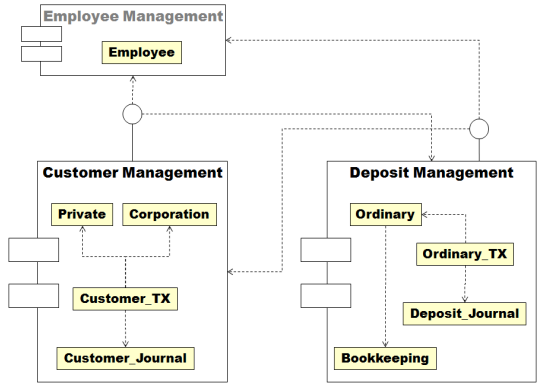
In order to measure complexity, reusability, customizability, we apply proposed metrics into several projects proceeded in the banking domain. The reason is that because various components for the same purpose may be developed in the same domain, we may measure the complexity, customizability, and reusability of components. In this chapter, we demonstrate the measurement results by applying metrics into component design and implementation. Also, we discuss the difference of between existing metrics and proposed metrics.

We will estimate the costs and effort for component development or component-based software development through measurement results obtained using the previous metrics. Furthermore, we measure the component's quality when we register developed components in component repository.

4.1 Measurement Results of Complexity

In order to measure the complexity of each component in component design time, we should first develop component diagram. We apply proposed metrics into component diagram for banking domain. An example of component diagram is shown in [Figure 2]. The Figure 2 shows a part of banking component diagram such as customer management, employee management, and deposit management of banking domain.

As shown in Figure 2, there are three components: 'Customer Management', 'Employment Management', and 'Deposit Management'. Also, there are one or more classes in the each component. We measure the complexity of each component by using proposed CPC and CSC. Also, CDC is measured by using sequence diagram for each component.



[Fig. 2] Component Diagram

For example, the CPC and CSC of 'Customer Management' and 'Deposit Management' are measured with:

CPC(Customer Management) = 47 + 66 + 13 = 126, where,

CmpC= 10 + 1+36 = 47,

$\sum_{i=1}^{m} CC_i$ =30 + 9*4=66, and

$\sum_{j=1}^{n} MC_j$ =5+2*4=13.

CSC(Customer Management)=

(0*2) +(1*4)+(4*6)+(0*8) +(0*10) = 28.

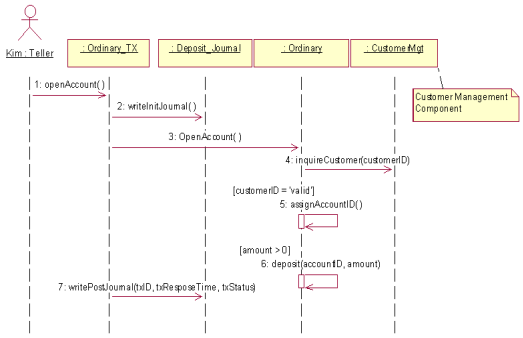
We give weight values for each relationship based on weight value table for relationships shown in Table 1.

[Table 1] Weight Value Table for Relationship

Relationships	Weight Values
Dependency	2
Association	4
Generalization	6
Aggregation	8
Composition	10

In order to measure the CDC of each component, we use sequence diagrams per a use case. An example of sequence diagrams is shown in Figure 3. Figure 3 shows the interactions among classes existent in a use case. The 'OpenAccount()' is the interface method declared in the interface of 'Deposit Management'.

As depicted in Figure 3, there are several message flows among classes contained in a component such as deposit component. It is difficult to measure the complexity of 'Deposit Management' component with only class diagrams or component diagrams.



[Fig. 3] 'Open Account' Sequence Diagram of Deposit Management

Therefore, it is enable to measure the dynamic complexity of each component with interaction diagrams. Applied CDC into this example diagram, the measurement result is given by:

DC(OpenAccount())=1+1+1+2+1+3+4=13.

[Table 2] Measurements of Interface Methods

Interface Methods	Deposit Management
OpenAccount	13
CloseAccount	20
InquireAccount	9
InquireTransactionHistory	13
InquireCustomerAccount	9
Deposit	22
WithDraw	24
Transfer	17
InquireBookKeeping	11
AssignAccountId	2
CallInterest	5

Also we calculate the value of CCC for each component by using lines of code. We developed each component in forms of EJB Beans. Therefore, we measure the CCC of each component by combining CPC and cyclomatic complexity. The results are as following:

CCC(Customer Management) = 47 + 66 + 13 + 98 = 224.

CCC(Deposit Management) = 114 + 69 + 68 + 70 = 321.

Here we calculate the CCM by applying cyclomatic

complexity of each method. After the CCM of each method in each class contained in a component is calculated, the summation of values of each CCM becomes CCM of 'Customer Management' component. The resulting value is 98. The values of CmpC, the sum of class complexity, and the sum of method complexity are equal to the values of CPC.

We have learned that the measurement results of CCC are larger than of CPC. It means that the larger the complexity of CPC, the larger the complexity of CCC.

4.3 Measurement Results of Customizability

In order to measure the customizability of each component, we use the component specification for each component specification. During the component analysis, we identified commonality and variability of components will be developed in a domain. Then, identified variability methods are described as customization methods in component specifications at design phase. Customization methods of 'Customer Management' and 'Deposit Management' are described in Table 3.

We measure the customizability of 'Customer Management' component and 'Deposit Management' component by using CV metric. The results obtained are given by:

CV(Customer Management) = 2/11≈0.18.

CV(Deposit Management) = 2/11≈0.18.

For example, there are two customization methods in the 'Deposit Management' Component. Therefore, the CV(Deposit Management) may be calculated by dividing customization methods into total interface methods.

[Table 3] Customization Methods

IcustomerManagement	IdepositManagement
SetCorporationID (corporationIDFlag): Boolean	SetAccountFlag (accountFlag):Boolean
SetCustomerIDFlag (customerIDFlag):Boolean	SetInterestFlag (interestFlag1, interestFlag2):Boolean

4.4 Measurement Results of Reusability

We measure the reusability by using CRLFunc and CRLLOCs. CRLFunc is applied into designed components, while CRLLOCs is applied into implemented applications because CRLLOCs measures the percentage how many parts of a component is reused in an application. For example, CRLFunc of 'Deposit Management' component and 'Customer Management' component is obtained with:

CRLFunc (Customer Management)= 9/9=1

CRLFunc (Deposit Management)= 9/11≈0.819

We developed component-based banking systems by using 'Customer Management', 'Deposit Management', and 'Employee Management'. Then we measure the reuse level of each component in banking system development through lines of code.

The measurement results of each component are given by:

CRLLOCs (Customer Management)= 34/576*100% ≈ 5.9%

CRLLOCs (Deposit Management)= 28/576*100% ≈ 4.9%

4.5 Assessment

In this section, we discuss different metrics proposed in this paper to measure component's quality and their pros and cons. Table 4 lists approaches and factors to measure component's quality.

[Table 4] Comparisons of Different Metrics

	CPC	CSC	CDC	CCC	CV	CR	CRL_{LOC}
Class	O			O			O
Interface	O			O			O
Class Method	O		O	O			O
Interface Method	O		O	O	O	O	O
Attributes	O			O			O
Parameters	O		O	O			O
Relationship		O					
Messages			O				O
Cyclomatic complexity				O			
Customization Methods				O	O		O
Common Method				O	O	O	O
Lines of Code				O			O

As shown in Table 4, the number of factors of metrics applied in design time is fewer than the number of factors

of metrics applied in implementation time. Therefore, measurement results of complexity or reusability by using CCC and CRLLOCs are more accurate than of by using CPC, CSC, CDC, and CR.

However, we estimate the size of component, costs, or efforts required in component development or component-based software development because CPC, CSC, CDC, and CR may be measured early in CBD.

[Table 5] Component-Oriented Metrics Effects

Metrics	Objective	C.T.E	Und	Main	C.D.E	CBSD.E	Cust
CPC	↓	↓	↑	↑	↓	↓	
CSC	↓	↓	↑	↑	↓		
CDC	↓	↓	↑	↑	↓		
CCC	↓	↓	↑	↑	↓		
CV	↑	↑	↓	↑	↑	↓	↑
CRL_{Func}	↑		↑	↑		↓	↑
CRL_{LOC}	↑		↑	↑		↓	↑

*C.T.E: Component Testing Efforts, Und: Understandability, Main: Maintainability, C.D.E: Component Development Effort, CBSD.E: CBSD Effort, Cust: Customizability

Proposed component-oriented metrics help evaluate the development and testing efforts needed, understandability, maintainability, and reusability. This information is summarized in Table 5.

Proposed component-oriented metrics provide valuable information to component developers, component assemblers, application developers and project managers.

5. Concluding Remarks

In this paper, we have measured the complexity, customizability, and reusability of components produced during component development process for banking domain. Several different metrics have been for this purpose, CPC, CSC, CDC, CCC, CV, CR, and CRL. Especially we applied CRL to measure the reuse level of developed components into component-based banking systems.

We have found that the complexity of a component may help to estimate the component's size. Also, reusability and customizability of components effect on the reusability of components during component based software development.

Finally, we have found that lines of code of components are suitable for measurements of reusability in CBSD. However, we do not consider the complexity of technical complexity of each component. We will expect that the complexity and reusability of components may be calculated by using function points. Traditional function points are not suitable in component based software development. We will research the component-oriented function points and complexity metrics

References

[1] Szyperski C., Component Software: Beyound Object-Oriented Programming, Addison Wesley Longman, Reading, Mass., 1998.

[2] Linda H. Rosenberg, "Applying and Interpreting Object-Oriented Metrics", at URL: http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html.

[3] Sun Microsystems Inc., "Enterprise JavaBeans Specifications", at URL: http://www.javasoft.com

[4] Rational Software Corp., Unified Modeling Language(UML) Summary, 1997.

[5] Object Management Group, "CORBA Components", at UR: http://www.omg.org, March 1999.

[6] Norman E. Fenton and Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Company, 1997.

[7] Chidamber, Shyam and Kemerer, Chris, "A metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, June, 1994, pp. 476-492.

[8] Lorenz, Mark and Kidd, Jeff, Object Oriented Software Metrics, Prentice Hall Publishing, 1994.

[9] McCabe & Associates, McCabe Object Oriented Tool User's Instructions, 1994.

[10] Rosenberg, Ian, "Metrics for Object Oriented Environments", EFAITP/AIE Third Annual Software Metrics conference, December, 1997.

[11] Hudli, R., Hoskins, C., Hudli, A., "Software Metrics for Object Oriented Designs", IEEE, 1994.

[12] 한정수, "컴포넌트 재사용을 위한 효율적인 사용자 검색 피드백에 관한 연구", 한국산학기술학회논문지, Vol.7, No. 3, pp.379-384, 2006년 3월.

[13] 궁상환, "품질속성을 고려한 소프트웨어 아키텍처 패턴의 정의", 한국산학기술학회논문지, Vol.8, No.1,

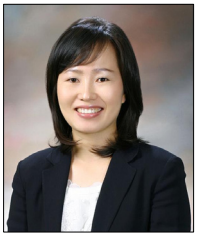
pp.82~95, 2007년 2월.

Chul-Jin Kim [Regular member]

- Feb. 1996 : Kyonggi Univ., B.E.
- Feb. 1998 : Soongsil Univ., M.S
- Feb. 2004 : Soongsil Univ., Ph.D
- Sept.2004 : Catholic Univ. Visiting Professor
- Dec. 2004 ~ Dec. 2009 : Samsung Electronics Co.
- Mar.2009 ~ current : Inha Technical College, Dept of Computer System, Assistant Professor

<Research Interests>
CBD, Component Customization, Embedded Software

Eun-Sook Cho [Regular member]



- Feb. 1993 : Dongeui Univ. B.E
- Feb. 1996 : Soongsil Univ., M.S
- Feb. 2000 : Soongsil Univ., Ph.D
- Jan. 2002 ~ Oct. 2003 : Invited Researcher in ETR
- Sep. 2000 ~ Feb. 2005 : Dongduk Women's Univ., Dept of Data Information, Full-time Instructor
- Mar. 2005 ~ current : Seoil Univ., Dept of Software, Assistant Professor

<Research Interests>
CBSE, Embedded Software, Service- Oriented Computing, SOA