A Metrics Suite for Measuring Reusability of Software Components

Hironori Washizaki[1], Hirokazu Yamamoto[2] and Yoshiaki Fukazawa[1]
[1]*Department of Computer Science, Waseda University*
3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
{ *washi, fukazawa* }*@fuka.info.waseda.ac.jp*
[2]*Matsushita Electric Industrial Co., Ltd.*
1006 Kadoma, Kadoma City, Osaka 571-8501, Japan
h-yama@isl.mei.co.jp

Abstract

In component-based software development, it is necessary to measure the reusability of components in order to realize the reuse of components effectively. There are some product metrics for measuring the reusability of Object-Oriented software. However, in application development with reuse, it is difficult to use conventional metrics because the source codes of components cannot be obtained, and these metrics require analysis of source codes. In this paper, we propose a metrics suite for measuring the reusability of such black-box components based on limited information that can be obtained from the outside of components without any source codes. We define five metrics for measuring a component's understandability, adaptability, and portability, with confidence intervals that were set by statistical analysis of a number of JavaBeans components. Moreover, we provide a reusability metric by combining these metrics based on a reusability model. As a result of evaluation experiments, it is found that our metrics can effectively identify black-box components with high reusability.

1. Introduction

Component-based software development (CBD) has become widely accepted as a cost-effective approach to software development, as it emphasizes the design and construction of software systems using reusable components[1]. CBD is capable of reducing developmental costs and improving the reliability of an entire software system. Since it is natural to model and implement components in an object-oriented paradigm/language[2], we limit this study to the use of OO language for the implementation of components. In a narrow sense, a software component is defined as a unit of composition. A user of a component can exchange it independently in the form of an object code without source codes. We call such components, whose source codes are unavailable for users, "black-box components."

As the number of components available on the market increases, it is becoming more important to devise software metrics to qualify the various characteristics of components. Software metrics are intended to measure software quality characteristics quantitatively. Among several quality characteristics, the reusability is particularly important when reusing components. It is necessary to measure the reusability of components in order to realize the reuse of components effectively.

Some measurement methods (product metrics) for measuring the reusability of OO software, such as Chidamber and Kemerer's metrics[3](C&K metrics), have been proposed[4]. However, since many of these conventional metrics require analysis of source codes, these metrics cannot be applied to black-box components. Moreover, conventional metrics do not consider the features of components. CBD requires a considerably different approach to that of OO methods.

In this paper, we propose a metrics suite for measuring the reusability of black-box components for the activity of development with component reuse, based on the limited static information that can be obtained from the outside of components without any source codes, in order to identify the best components in terms of their reusability.

2. Software components

A software component is a unit of composition with contractually specified interfaces. The component can be used from the outside of it via the interfaces. To component users, a component is a self-contained unit that can be used for a specific purpose. The internal implementation of a

Proceedings of the Ninth International Software Metrics Symposium (METRICS'03)
1530-1435/03 $17.00 © 2003 **IEEE**

component is usually hidden from its users. Components are not only reused within organizations to which the components' developers belong, but are also distributed in the form of an object code via the Internet and reused in other environments[5]. Therefore, users who want to reuse components often cannot obtain source codes of the components except for object codes.

2.1. Granularity of component

The granularity of the component can be defined as the conceptual size of the component's functions. The granularities of components are classified into the following: coarse-grained, medium-grained and fine-grained[6].

The business component that encapsulates business logic is a coarse-grained component. The application component, which is composed of fine-grained components and specific logic, is a medium-grained component.

GUI widgets and generic components with minimum logic are fine-grained components. These fine-grained components are mainly provided by the component library attached to the Rapid Application Development (RAD) tool.

2.2. Fine-grained components

Currently, fine-grained components are the most widespread due to the success of the RAD tool. ActiveX[7] and JavaBeans[8] are component architectures that are suitable for treating fine-grained components. In this paper, the target component architecture is JavaBeans, which brings the component development model to Java. However, our approach can be similarly applied to other component architectures that are suitable for treating fine-grained components.

A JavaBeans component is a reusable component that can be manipulated visually in a RAD tool[8]. One JavaBeans component is composed of one or more Java classes, and opens only one Facade class to the public, based on the Facade pattern[9]. The Facade class is a front class for the outside of a component, and all method invocations from outside of the component can be realized via the Facade class. The name of the Facade class becomes the name of the corresponding component.

There are four important features of any JavaBeans component.

- Property: Properties are the named attributes associated with a JavaBeans component, whose values can be read or written by invoking appropriate read/write methods. Usually, properties correspond to the Facade class's fields one-to-one. Properties whose values can be read are called "readable properties", and those whose values can be written are called "writable properties."

- Read method: Read methods are methods implemented within the Facade class to read the properties' values from outside of the component.

- Write method: Write methods are methods implemented within the Facade class to change the properties' values from outside of the component.

- Business method: Business methods are simply normal Java methods that can be invoked from outside of the component, except for write/read methods, implemented within the Facade class.

Information on the number and type of the above-mentioned features can be statically obtained for all black-box JavaBeans components without source codes by using Java's introspection mechanism, In addition to these features, information on the number and type of fields of the Facade class can also be statically obtained for all JavaBeans components.

JavaBeans provides naming conventions that enable JavaBeans components to be manipulated in a uniform way, particularly with respect to the component properties[1]. Using the introspection mechanism that JavaBeans provides, the method whose name is "setXXX" or "getXXX" can be recognized as the write method or the read method corresponding to the property "XXX".

Moreover, by preparing BeanInfo classes that represent the meta-information of components, developers of components can specify the read/write methods and properties without using the naming conventions. BeanInfo classes are classes that implement the BeanInfo interface that is provided by the Java Development Kit APIs.

Figure 1 shows the UML class diagram[10] of a typical JavaBeans component. In Figure 1, "Stock" is composed of three classes, and has four fields ("amount", "brand", a reference pointer to an instance of "Broker", and a reference pointer to an instance of "Market") in the facade class. The Facade class of the component is the class "Stock." Since there are a read method "getBrand" and a write method "setBrand" corresponding to the field, the introspection mechanism recognizes that the component also has one readable and writable property, named "brand."

In the following section, we will define a reusability model for black-box components, and consider the reusability metrics for black-box components in terms of their properties, read methods, write methods, and business methods.

3. Reusability metrics for components

CBD is characterized by two activities: development of components for reuse, and development of software systems with reuse of components[11]. The activity of development

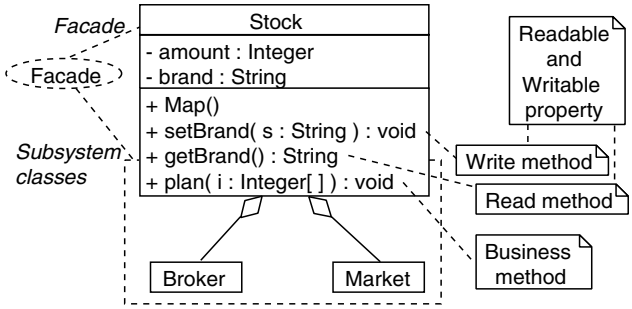

Figure 1. Class diagram of a JavaBeans component

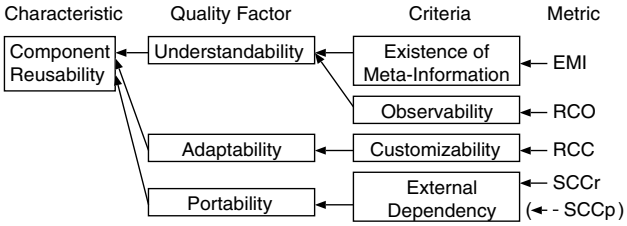

Figure 2. Black-box component reusability model

with reuse is realized by developing software with existing components.

For users who want to reuse components in the activity of development with reuse, one of the most important quality characteristics of a component is its reusability. Reusability is the degree to which a component can be reused, and reduces the software development cost by enabling less writing and more assembly. How users can detect which component is the most reusable among several components implementing the same specification, and how users can select components with higher reusability are key issues. It is necessary to measure the reusability of components in order to realize the reuse of components effectively.

3.1. Component reusability model

We define a new reusability model for black-box components from the viewpoint of component users. Figure 2 shows our reusability model based on the McCall's Factor-Criteria-Metrics (FCM[12]) approach. The FCM approach is used by ISO; for example, ISO 9126[13] applies the FCM model to the assessment for software quality. Our reusability model is inspired from the reusability model given by the REBOOT approach[14], but the criteria and metrics used in our model are quite different from those in the REBOOT approach. The measurement target of our model is also different from that of the REBOOT approach. The reusability model is structured in three levels in a top-down manner. The three terms (factor, criterion and metric) are organized in an evaluation and assessment hierarchy. Factors are used at management level, as the nonfunctional requirements of a component. Criteria are used at (application) design level. Metrics are used at product level.

Metrics are used to determine quality factors that affect reusability. A component alone has certain characteristics that tend to affect its reusability. In this reusability model, the component reusability can be decomposed into three factors: understandability, adaptability, and portability. This reusability model includes aspects related to the Understandability, Adaptability, and Portability factors given by ISO 9126[13]. The quality factors are selected only to provide an analysis of the reusability of a component. Factors related to other aspects of component quality that are not considered to be important to reusability are not considered. The choice of the three factors affecting reusability has been made on the basis of an analysis of the activities carried out when reusing a black-box component. These activities are as follows.

- Understanding the functionality of the component, to decide whether it meets the new functional requirements.

  A user needs high understandability to do this activity. Understandability is defined based on the estimated effort needed by a user to recognize the concept behind a component and its applicability.

- Adapting the component to the specific functional requirements of the new system.

  A user needs high adaptability to do this activity. Adaptability is the ease with which a component can be adapted to fulfill a requirement that differs from that for which it was originally developed.

- Porting the component to a new environment.

  A user needs high portability to do this activity. Portability is the ease with which software can be transferred from one environment to another.

These factors are hierarchically subdivided into criterion and metrics as shown in Figure 2.

There are well-known reusability criterion that do not appear in our model: cohesion and coupling. We do not use these criterion because these criterion usually need metrics that require the analysis of source codes of components (described in Section 6).

A brief description of the criteria in our model and of the related evaluation is given below.

(1) Existence of meta-information

The existence of meta-information indicates whether the meta-information of the target component is provided. If the meta-information of the component is provided, users of the component can easily understand the component's usage, which the component's developer assumes.

We can specify the existence of meta-information by checking whether the BeanInfo class corresponding to the target component is provided.

(2) Observability

Observability measures how easy it is to observe a component in terms of its operational behaviors, input parameters, and outputs.

The behavior of a black-box component can be mainly confirmed by using read methods corresponding to the component properties. Therefore, the number of read methods provided by a component is the important factor contributing to the observability of the component.

JavaBeans allows software developers to construct applications by piecing components together either programmatically or visually. Thus, JavaBeans components must allow their property values to be observed programmatically or through some type of visual interface. Therefore, many components expose fields of Facade classes intrinsic to their behavior as readable properties by the read methods.

Figure 3 shows an example of using BeanBox[15] as a RAD tool that provides visual interfaces to manipulate JavaBeans components visually. A line chart component, named "FukaGraphBean"[16], is selected on BeanBox, and BeanBox displays all of the properties of the component on a property sheet located at the right-hand side of Figure 3. For example, by seeing the property sheet, a user can confirm the value of the property, named "colorGraph", that determines the color of the graph.

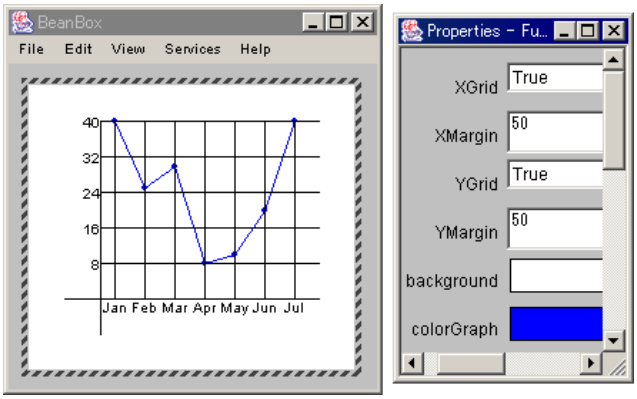

Figure 3. Example of using RAD tool

(3) Customizability

Customizability indicates the built-in capability for supporting the customization and configuration of a component's internal functional features.

Properties are used for component customization and configuration[17]. Therefore, components must allow their property values to be changed programmatically or through some type of visual interface for their customization. The number of write methods provided by a component strongly affects the customizability of the component.

(4) External dependency

External dependency indicates the component's degree of independence from the rest of the software which originally used the component.

Invoking component's business methods from the outside of the component provides functions to outside. Therefore, the external dependency of the business methods implemented within a component strongly affects the component's external dependency. In business methods, there is a possibility that parameters or return values depend on the rest of the software which originally used the component. In contrast, business methods without parameters or return values provide functions that are self-completed within the component.

3.2. Definitions of reusability metrics

According to the above-mentioned reusability model, we define five metrics, EMI, RCO, RCC, SCCr, and SCCp, for measuring the existence of meta-information, observability, customizability, and external dependency of a black-box JavaBeans component, with confidence limits that were set by statistical analysis of a number of components (shown in Section 4).

Measurement values of all our metrics are always normalized to a number between 0 and 1. Each metric has a confidence interval[18] with confidence coefficient 95% (except the upper confidence limit of SCCr). If the value of each metric M is in a confidence interval $[LL_M, UL_M]$ from the lower confidence limit LL_M to the upper confidence limit UL_M, the quality factor corresponding to the metric is thought to be appropriately high.

3.2.1 Existence of Meta-Information

Definition 3.1 (EMI: Existence of Meta-Information)

$EMI(c)$ is whether the BeanInfo class corresponding to the target component c is provided:

$$EMI(c) = \begin{cases} 1 & \text{(BeanInfo class exists)} \\ 0 & \text{(otherwise)} \end{cases}$$

Confidence interval: [0.5, 1.0]

Consideration: If the value of $EMI(c)$ is 1, users of c can easily understand the usage of c, which c's developer assumes.

Example: Figure 4 shows a comparison of measurement values in terms of our five metrics for two components (Chart and Graph). In Figure 4, Chart has a BeanInfo class (ChartBeanInfo), and Graph does not. In this case, the understandability of Chart is thought to be higher than that of Graph.

3.2.2 Rate of Component Observability

Definition 3.2 (RCO: Rate of Component Observability)

$RCO(c)$ is a percentage of readable properties in all fields implemented within the Facade class of a component c.

$$RCO(c) = \begin{cases} \frac{P_r(c)}{A(c)} & (A(c) > 0) \\ 0 & (\text{otherwise}) \end{cases}$$

where:
$P_r(c)$: number of readable properties in c
$A(c)$: number of fields in c's Facade class

Confidence interval: [0.17, 0.42]
Consideration: RCO indicates the component's degree of observability for users of the component. To understand the behavior of a component from outside the component, the observability of the component should be high. However, there is a possibility that it is difficult for users to find an important readable property among all of the readable properties when the observability is too high.

If the value of $RCO(c)$ is in the confidence interval, the height of the observability is appropriate, and the understandability of c is high.
Example: In Figure 4, Chart's Facade class has four fields ("title", "values", a reference pointer to an instance of Grid, and a reference pointer to an instance of Border), and Chart has one read method (getTitle) corresponding to the field ("title"). Graph's Facade class also has four fields, but Graph has no read methods. In this case, the understandability of Chart is higher than that of Graph.

3.2.3 Rate of Component Customizability

Definition 3.3 (RCC: Rate of Component Customizability)

$RCC(c)$ is a percentage of writable properties in all fields implemented within a Facade class of a component c:

$$RCC(c) = \begin{cases} \frac{P_w(c)}{A(c)} & (A(c) > 0) \\ 0 & (\text{otherwise}) \end{cases}$$

where:
$P_w(c)$: number of writable properties in c

Confidence interval: [0.17, 0.34]
Consideration: RCC indicates the component's degree of customizability for users of the component. To adapt the settings of a component from outside the component to the user's requirements, the customizability of the component should be high. However, too high a customizability violates the encapsulation of the component, and leads to larger opportunities of wrong use.

If the value of $RCC(c)$ is in the confidence interval, the height of the customizability is appropriate, and the adaptability of c is high.
Example: In Figure 4, Chart's Facade class has four fields, and Chart has one write method (setTitle) corresponding to the field. Graph's Facade class also has four fields, but Graph has no write methods. In this case, the adaptability of Chart is higher than that of Graph.

3.2.4 Self-Completeness of Component's Return Value

Definition 3.4 (SCCr: Self-Completeness of Component's Return Value)

$SCCr(c)$ is a percentage of business methods without any return value in all business methods implemented within a component c:

$$SCCr(c) = \begin{cases} \frac{B_v(c)}{B(c)} & (B(c) > 0) \\ 1 & (\text{otherwise}) \end{cases}$$

where:
$B_v(c)$: number of business methods without return value in c
$B(c)$: number of business methods in c

Confidence interval: [0.61, 1.0]

We first obtained 0.96 as the upper confidence limit of SCCr (UL_{SCCr}) by statistical analysis. However, we found that components tend to have business methods without return values regardless of the reusability (described in Section 4.4). Therefore, we use 1.0 as UL_{SCCr}.
Consideration: SCCr indicates the component's degree of self-completeness, and the low degree of external dependency for users of the component. Simply, the smaller the number of business methods without return value, the smaller the possibility of the component having external dependency. High self-completeness of a component (low external dependency) leads to high portability of the component.

If the value of $SCCr(c)$ is in the confidence interval, the external dependency to outside of a component is low, and the portability of c is high.
Example: In Figure 4, Chart provides two business methods (plot, grid), and the types of these business methods' return values are $void$. Since $void$ represents no value in

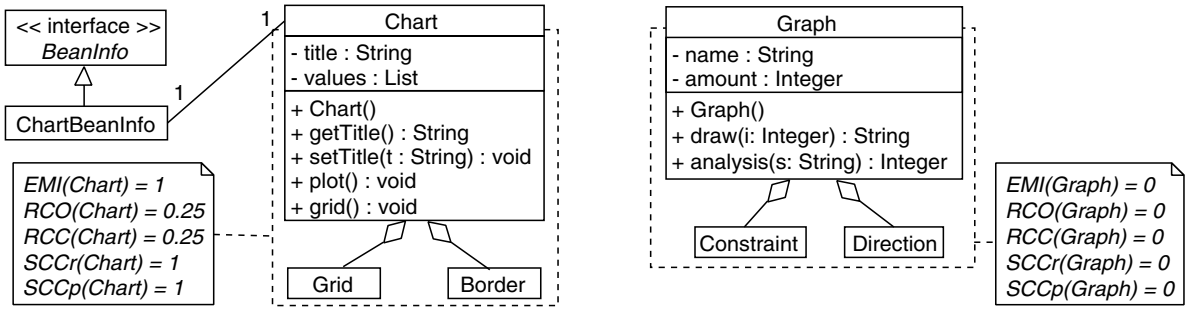

Figure 4. Example of a comparison of measurement values

Java language, these business methods are those without return values. Graph has two business methods with specific types of return values. In this case, the portability of Chart is higher than that of Graph.

3.2.5 Self-Completeness of Component's Parameter

Definition 3.5 (SCCp: Self-Completeness of Component's Parameter)

$SCCp(c)$ is the percentage of business methods without any parameters in all business methods implemented within a component c:

$$SCCp(c) = \begin{cases} \frac{B_p(c)}{B(c)} & (B(c) > 0) \\ 1 & (\text{otherwise}) \end{cases}$$

where:
$B_p(c)$: number of business methods without parameters in c

Confidence interval: [0.42, 0.77]
Consideration: SCCp indicates the component's degree of self-completeness, and the low degree of external dependency for users of the component. Simply, the smaller the number of business methods without parameters, the smaller the possibility of having dependency outside the component.
Example: In Figure 4, Chart provides two business methods (plot, grid), and these business methods have no parameters. Graph has two business methods with parameters. In this case, the portability of Chart is thought to be higher than that of Graph.

4. Validation of reusability metrics

We have developed a component analysis tool for black-box JavaBeans components in Java language. Our tool automatically measures primitive values, such as the number of methods in a Facade class, and calculates the values of our five metrics using the measured primitive values.

Metrics without any thresholds or appropriate intervals for measurement values cannot be effectively used. We calculated the confidence intervals of our four metrics by using 125 JavaBeans components provided at JARS.COM[19] and our analysis tool.

At JARS.COM, a rating committee reviews all submitted components, and rates all components by giving rating scores from 0.125 to 1.0 (eight levels) based on the overall quality of components. In the following, the rating score of a component c is described as $J(c)$. Submitted components are reviewed and evaluated based on presentation, functionality, and originality:

- Presentation: Look & feel and documentation
- Functionality: Usefulness and ease of use,
- Originality: Uniqueness and innovativity.

Although the rating score does not directly indicate the reusability of a component, the above-mentioned characteristics that are considered in the review (presentation, functionality, and originality) tend to affect the component's reusability. Therefore, we believe the rating score reflects the reusability of a component, and also reflects quality factors that impact on the reusability (understandability, adaptability, and portability).

The rating committee also categorizes components into nine categories. Table 1 shows the number of components and the average of the rating scores in each category among all 125 samples. The average of the numbers of properties among all samples is 5.14, and the average of the numbers of business methods is 8.23.

We calculated the confidence intervals of our metrics in terms of the reusable components using the rating scores. First, we assumed that the reusability of components ($S = \{c_1, ..., c_n\}$) that satisfy $J(c_i) \geq 0.875$ is high. Second, by an interval estimation based on the inversion of the one-sample Wilcoxon signed-rank (nonparametric) test statistic[18], we calculated confidence intervals with confidence coefficient 95% for our five metrics (except the upper

Table 1. Number of components and $J(c)$

Category	Number of components	Average of $J(c)$
Programming	98	0.42
WWW	4	0.71
Game	3	0.79
Utilities	4	0.71
Science	16	0.79
Total	125	0.50

confidence limit of SCCr) using all components that satisfy $J(c_i) \geq 0.875$ from among all samples. The number of such high rated components was 19.

Table 2 lists the lower confidence limit LL_M and upper confidence limit UL_M for each metric M of our metrics. Table 2 also lists the average of measurement values of all samples, and the number of components whose measurement values are in the confidence interval (Corresponding components). In the following, the measurement value of the metric M for the component c is denoted as $M(c)$.

Table 2. Average of $M(c)$, etc.

Metric	All average of $M(c)$	LL_M	UL_M	Corresponding components
EMI	0.84	0.5	1.0	105
RCO	0.40	0.17	0.42	36
RCC	0.35	0.17	0.34	35
SCCr	0.85	0.61	1.0	108
SCCp	0.74	0.42	0.77	28

4.1. Measurement result of EMI

We categorized all samples into two categories ($J(c) < 0.875$ and $J(c) \geq 0.875$), and counted the number of components that satisfy $EMI(c) = 1$. Table 3 lists the results of the number of components.

In Table 3, 84% of all samples have BeanInfo classes. Regardless of the height of rating scores, components tend to have BeanInfo classes. Moreover, 68% of components that satisfy $J(c) \geq 0.875$ have BeanInfo classes. Therefore, the value of EMI should be 1 for high understandability of components, but it is difficult to judge the reusability using only the value of EMI.

4.2. Measurement result of RCO

Figure 5 shows the number of components in each section among ten evenly split sections with respect to measurement values of RCO. Figure 5 also shows averages of rating scores in each section.

Table 3. Number of component in categories

EMI	Number of components	
	$J(c) < 0.875$	$J(c) \geq 0.875$
1	92	13
0	14	6

From the obtained confidence interval, the understandability of components that expose $1/5 \sim 2/5$ of the Facade classes' fields as readable properties by read methods is high.

Among all samples, the percentage of components whose values of RCO are in the confidence interval is only 29% (shown in Table 2). This means that the confidence interval of RCO is not a general range where every component corresponds. Therefore, it is evident that RCO with its confidence interval can be effectively used for measuring the understandability and reusability of components.

Moreover, the rating scores of components whose values of RCO are higher than 0.8 tend to be low. This means that components with too high an observability are not reusable; the fields of components' Facade classes should be hidden to some extent.

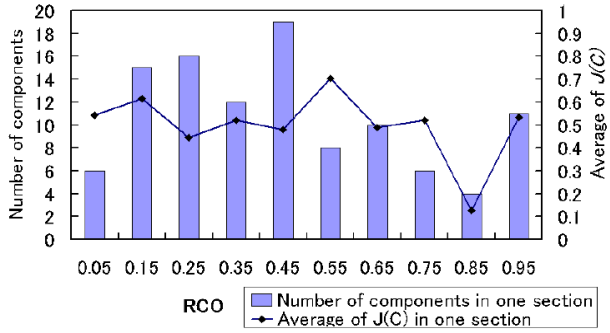

Figure 5. Histograms for RCO

4.3. Measurement result of RCC

Figure 6 shows the histograms of measured values of RCC. From the obtained confidence interval, the adaptability of components that expose $1/5 \sim 1/3$ of the Facade classes' fields as writable properties by write methods is high.

In all samples, the percentage of components whose values of RCC are in the confidence interval is 28% (shown in Table 2). This means that the confidence interval of RCC is not a general range where every component corresponds. Therefore, it is found that RCC with its confidence interval

can be effectively used for measuring the adaptability and reusability of components.

Moreover, the rating scores of components whose values of RCC are higher than 0.6 tend to be low. This means that components with too high a customizability are not reusable.

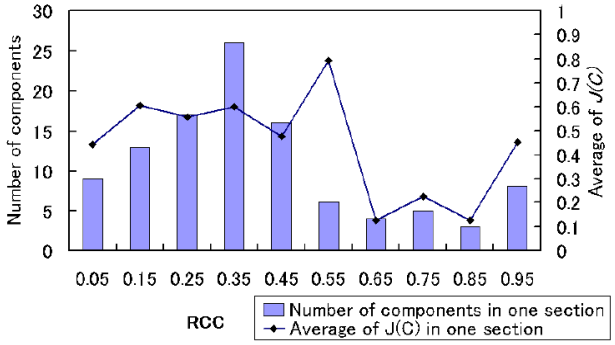

Figure 6. Histograms for RCC

4.4. Measurement result of SCCr

Figure 7 shows the histograms of measured values of SCCr. From the obtained confidence interval, the portability of components for which more than $2/3$ of the business methods have no return values is high. This result is related to the typical usage of the component. In general, a procedure for utilizing a component's functions is composed of the following three steps:

(1) Set values into writable properties by using write methods to input user's values.

(2) Invoke some business methods to use provided functions. Results of the functions may by stored in the values of properties.

(3) Get values of readable properties by read methods to obtain the results of executed functions.

The result of measuring SCCr means it is preferable that a user of a component obtains the results of the business method's invocation by invoking the read methods of readable properties, rather than by capturing the return value of the business method.

In all samples, the percentage of components whose values of SCCr are in the confidence interval is 86% (shown in Table 2). Moreover, in Figure 7, components tend to have business methods without return values regardless of the height of rating scores. Therefore, the value of SCCr should be more than $2/3$ for high portability of components, but it is difficult to judge the reusability using only the value of SCCr.

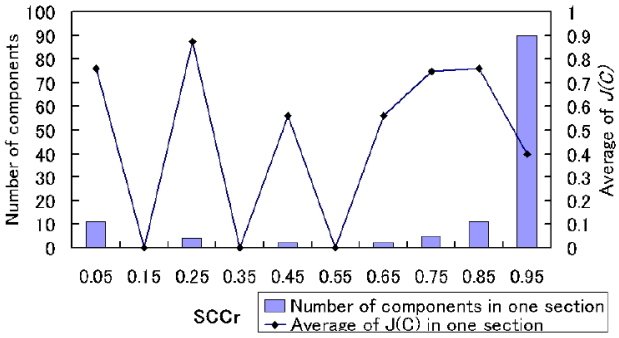

Figure 7. Histograms for SCCr

4.5. Measurement result of SCCp

Figure 8 shows the histograms of measured values of SCCp. In all samples, the percentage of components whose values of SCCp are in the confidence interval [0.42, 0.77] is 22% (shown in Table 2).

However, in Figure 8, components tend to have business methods without any parameters regardless of the height of rating scores. Therefore, it is difficult to judge the reusability using only the value of SCCp.

This result is related to the above-mentioned typical usage of the component. It is preferable that a user of a component inputs values to the component by the write methods of writable properties before invoking business methods, rather than by parameters of the business method.

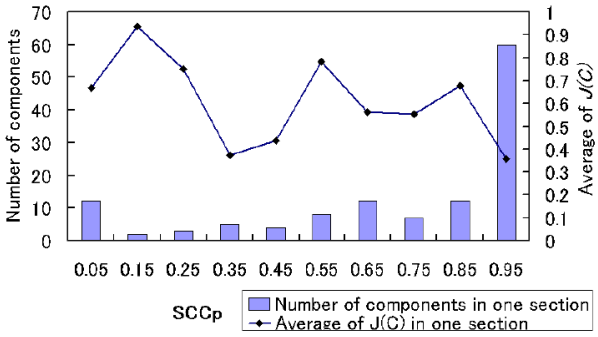

Figure 8. Histograms for SCCp

4.6. Correlation analysis

We verified correlations among our four metrics. Table 4 lists the coefficients of correlation for all combinations of measurement values of our four metrics (RCO, RCC, SCCr,

and SCCp). In Table 4, it is found that the metrics for the properties (RCO and RCC) are independent of the metrics for the business methods (SCCr and SCCp).

Table 4. Correlations coefficient table

Correlation coefficient	RCO	RCC	SCCr	SCCp
RCO	1	0.876	0.084	-0.037
RCC	0.876	1	0.151	-0.001
SCCr	0.084	0.151	1	0.516
SCCp	-0.037	-0.001	0.516	1

4.6.1 Correlation between RCO and RCC

Figure 9 shows a scatter diagram comparing the measurement values of RCO and those of RCC for all samples. The coefficient of correlation is $r = 0.876$, so there is a very strong positive correlation between RCO and RCC. This means that the developers of components ordinarily prepare both the read method and the write method corresponding to the same field of Facade classes when developers feel necessity of exposing the field.

In Section 4.2 and Section 4.3, the confidence interval of RCO is very similar to that of RCC. Therefore, if the understandability of a component were high, the adaptability would also be high.

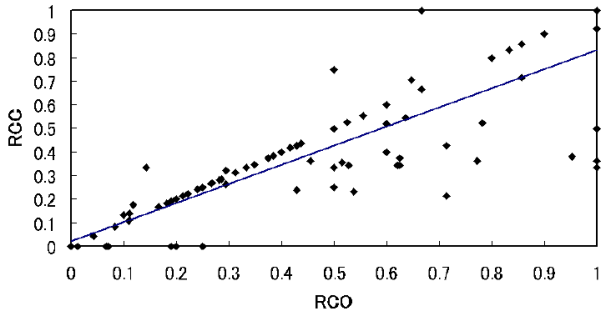

Figure 9. Scatter diagram of RCO and RCC

4.6.2 Correlation between SCCr and SCCp

Figure 10 shows a scatter diagram comparing the measurement values of SCCr and those of SCCp for all samples. The coefficient of correlation is $r = 0.516$; there is a positive correlation between SCCr and SCCp. This means that the developers of components often implement business methods without return values or parameters.

However, in Section 4.4 and Section 4.5, the confidence intervals of both metrics are completely different. This result originates in the fact that SCCp cannot reflect the portability of a component compared with SCCr. We believe that a user should use only SCCr for measuring the portability of a component.

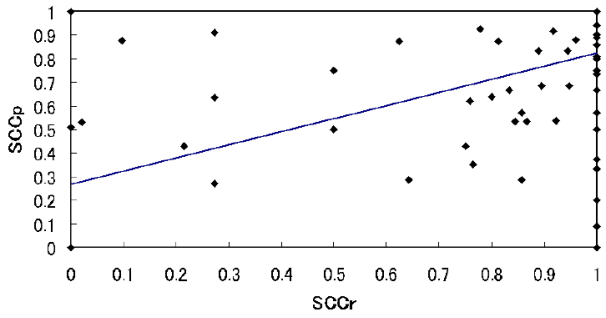

Figure 10. Scatter diagram of SCCr and SCCp

4.7. Metrics combination

Our five metrics can be used to judge the reusability by measuring detailed individual quality factors that are related to the reusability of a component. However, these metrics cannot measure an overall picture of the reusability in a simple manner.

Thus, we define a final reusability metric, named Component Overall Reusability (COR), by combining our metrics based on our reusability model. According to the reusability model, the reusability is composed of the understandability, adaptability, and portability.

We use only SCCr for the portability because SCCp is not appropriate as a metric for measuring the portability (described in Section 4.6.2).

Moreover, described in Section 4.6.1, the coefficient of correlation between RCO and RCC is 0.876. It is known that if there is a strong positive correlation between two metrics (correlation coefficient > 0.8), targeted quality factor can be measured enough using only one metric of these two metrics; if these two metrics are used together, one of these become redundant[20]. For users of a component, it is expected to specify the component's reusability using a fewer number of metrics. Therefore, since RCC is used for measuring customizability, we use only EMI for measuring understandability and do not use RCO.

We defined COR as a discriminant function that discriminates the reusable components that satisfy $J(c) \geq 0.875$ from other samples by a discriminant analysis based on the Stepwise method[21].

Definition 4.1 (COR: Component Overall Reusability)

$COR(c)$ indicates the component c's degree of reusability for users of the component.

$$COR(c) = 1.76\frac{V_{EMI}(c) + V_{RCC}(c) + V_{SCCr}(c)}{3} - 1.13$$

where:

$$V_M(c) = \begin{cases} 1 & (M(c) \geq LL_M \wedge M(c) \leq UL_M) \\ 0 & (\text{otherwise}) \end{cases}$$

Consideration: If the value of $COR(c)$ is larger than 0, the component c is thought to be reusable. Since the discriminant coefficient is 1.76, at least two of the component's understandability, adaptability, and portability should be high for users to reuse the component.

By using of COR, we have discriminated the reusable components with probability of 79% from other samples. Therefore, COR can be effectively used for the component selection in terms of reusability when there are several black-box components implementing the same specification.

5. Evaluation

We evaluated the usefulness of our three metrics, RCO, RCC, and COR. We verified the validity of the confidence intervals of these metrics by means of experimental tests.

We prepared three versions of FukaGraphBean[16] with different measurement values of RCO and RCC. FukaGraphBean is a line chart component, and originally has 13 fields within its Facade class, 13 read methods and 13 write methods corresponding to all fields. In the following, this original version of FukaGraphBean is described as C_A. Moreover, we prepared two other versions of FukaGraphBean (C_B and C_C). In C_B, several read methods and write methods that are unnecessary for the required functions of FukaGraphBean to be achieved are removed in order to put its values of RCO and RCC in the confidence intervals of their metrics. In C_C, all read methods and write methods that are unnecessary for the required functions to be achieved are removed.

Each version of FukaGraphBean is a black-box component without source codes for testers; testers cannot investigate the component's internal structure.

Table 5 lists the measurement values of RCO, RCC and COR for three versions of FukaGraphBean. The measurement values of C_B and C_C are in the confidence intervals in terms of RCO and RCC. Moreover, the measurement value of C_A is smaller than those of C_B and C_C in terms of COR. Therefore, the reusability of C_B and C_C is assumed to be higher than that of C_A.

Table 5. Three versions of FukaGraphBean

Version of FukaGraphBean	RCO	RCC	COR
C_A	1	1	0.04
C_B	0.38	0.31	0.63
C_C	0.23	0.23	0.63

Five testers constructed chart-drawing applications by piecing FukaGraphBean and other components (GUI text-input component, GUI label component, and GUI button component, etc.) together visually using BeanBox, with three versions of FukaGraphBean. FukaGraphBean provides almost all of the necessary functions which are required as application specifications.

For all tests, we measured the number of all executed operations, such as the operation of binding events to business methods by means of the mouse, operation of binding properties to other properties by means of the mouse, and the operation of changing a properties' values by means of the mouse and keyboard. When a tester only selects one of the properties or events and does not perform operation any more, we counted such incomplete operations as 0.5. Testers executed these operations to adapt all components and relations among them for application specifications.

Table 6 shows the comparison of the average for all testers of the execution time for constructing applications and the average of operation times on BeanBox. In Table 6, the number of operation times and the execution time of C_B are seen to be smaller than those of C_A. This result suggests that C_A lacks somewhat understandability and adaptability compared with C_B because testers could specify necessary readable/writable properties among limited properties in C_B much easier than in C_A.

The tester spent more time for C_B compared with that for C_C. This result originates in the fact that all of the necessary properties are already specified in C_C. However, the difference in execution times is not large, and the number of operation times of C_B is smaller than that of C_C. Therefore, it is found that C_B has high understandability while keeping an appropriately adaptability to meet the application's specifications, based on the confidence intervals of our metrics.

The understandability and adaptability of C_B and C_C, whose measurement values of RCO and RCC are in the confidence intervals, are more appropriate compared with those of C_A. Such appropriate understandability and adaptability lead to a high reusability of components. Since the measurement values of C_B and C_C are larger than that of C_A in terms of COR, it is confirmed that COR appropriately indicates the reusability of a component.

Table 6. Comparison of test results

Version	Number of executed operations	Execution time
C_A	18.1 [times]	274 [sec]
C_B	14.4 [times]	177 [sec]
C_C	16.2 [times]	152 [sec]

6. Related work

There are a significant number of software metrics for measuring a program size, related data structures and control structures. However, there are few metrics on the reusability of a software component in a CBD context. In the following, we compare the conventional reusability metrics with our metrics.

6.1. OO metrics

Some product metrics for measuring the reusability of OO software have been proposed. These metrics focus on the object structure which reflects the complexity of each individual entity, such as its methods and classes, and on the external complexity that measures the interactions among entities, such as coupling and inheritance.

Etzkorn's approach is to measure various static characteristics for C++ classes, such as the cohesion and complexity, and to add the measured values into one value that indicates the reusability of classes[22]. However, since this approach uses measured values that can be obtained by the analysis of source codes, it cannot be applied to black-box components.

C&K metrics is a suite of metrics for OO design, and is composed of six design metrics: Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object Classes (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM)[3]. Among these six metrics, WMC, DIT, CBO, and LCOM can be used to measure the reusability of OO classes[23].

However, CBO and LCOM cannot be applied to black-box components. CBO aims to measure the relations among classes by analyzing the source codes of the classes, and LCOM investigates the methods and fields in classes by analyzing the source codes.

WMC and DIT measure the characteristics of a single piece of a class without analyzing the source codes. Therefore, these can be applied to the Facade classes of black-box components. We measured WMC and DIT of all samples described in Section 4, and verified the correlations between measurement values and the rating scores. Definitions and measurement results of WMC and DIT are shown below.

6.1.1 Weighted Methods per Class (WMC)

WMC is a count of the methods implemented within a class or the sum of complexities of the methods. Since the method complexity cannot be measured for black-box components, we simply treat WMC as the methods count of the component's Facade class. The number of methods is a predictor of how much time and effort is required to develop and maintain the class.

Figure 11 shows the histograms of measured values of WMC. The confidence interval of WMC was [23, 50]. Among all samples, the percentage of components whose values of WMC are in the confidence interval is 22%.

While the measurement values of our metrics are always normalized to a number between 0 and 1, the value of WMC cannot be normalized. The obtained confidence interval of WMC greatly depends on what kinds of components are used, and may change in each measurement time. Moreover, WMC does not take into consideration the type of method (read, write, or business). Therefore, WMC cannot be efficiently applied to black-box components.

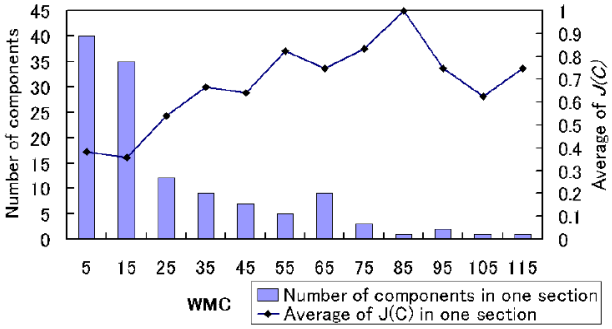

Figure 11. Histograms for WMC

6.1.2 Depth of Inheritance Tree (DIT)

DIT is the maximum number of steps from the class node to the root of the inheritance tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the more complex its behavior is to predict.

We measured DIT for the Facade class of the samples. Figure 12 shows the histograms of measured values of DIT. The confidence interval of DIT was [2, 4]. This result is similar to empirical idioms; most C++ classes in an application tend to be close to the root[3], while programmer-made Java classes should not be deeper than five levels from the root[24].

Among all samples, the percentage of components whose values of DIT are in the confidence interval is 71%. This means that the Facade classes of these components

tend to be shallower than five levels within the hierarchy, which is the same as normal classes. Therefore, it is difficult to judge the reusability by using only the value of DIT.

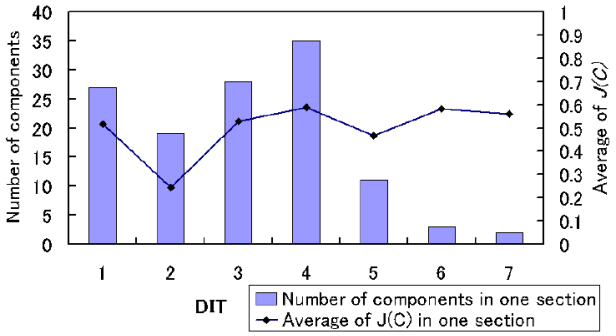

Figure 12. Histograms for DIT

6.2. Component metrics

There are some metrics for measuring a component's reusability. Cho proposes metrics for measuring the complexity, customizability, and reusability of software components[25]. The metrics require the analysis of source codes of components or the detailed analysis of components' specifications. Therefore, the metrics cannot be applied to black-box components which lack component specifications.

Wang proposes metrics for measuring the reusability of JavaBeans components[11]. The metrics indicate the actual reuse rates of the reused component in a component library and in a software product. However, the metrics cannot be used in a situation where sufficient time has not passed since the target component was developed. Moreover, since the metrics are based on the internal structure of the component, the metrics cannot be applied to black-box components.

In contrast, our metrics can be used in two situations where the source codes are unavailable and where the components were newly developed.

7. Conclusion

We proposed a metrics suite composed of six metrics for black-box components, and set confidence intervals for these metrics using the rating scores from JARS.COM. As a result of experimental evaluation, it is found that our metrics can be used to measure the component's reusability. Using our metrics with confidence intervals, users of components can easily select those with higher reusability. Therefore, our approach can help and promote the activity of development with the reuse of existing black-box components.

Our analysis tool is currently developed to accept JavaBeans components only. We will extend it to accept components based on other component architectures, such as ActiveX.

References

[1] C. Szyperski, Component Software: *Beyond Object-Oriented Programming*, Addison-Wesley, 1999.

[2] J. Hopkins, Component Primer, *Communications of the ACM*, Vol.43, No.10, ACM, 2000, pp. 27–30.

[3] S. Chidamber and C. Kemerer, A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering*, vol.20, No.6, IEEE CS, 1994, pp. 476–493.

[4] W. Frakes and C. Terry, Software Reuse: Metrics and Models, *ACM Computing Surveys*, Vol.28, No.2, ACM, 1996, pp. 415–435.

[5] M. Aoyama et al., Software Commerce Broker over the Internet, In *Proc. of 22nd Annual International Computer Software and Applications Conference*, IEEE CS, 1998, pp. 430–435.

[6] H. Washizaki and Y. Fukazawa, Dynamic Hierarchical Undo Facility in a Fine-Grained Component Environment, In *Proc. of 40th International Conference on Technology of Object-Oriented Languages and Systems*, Australian Computer Society, 2002, pp. 191–199.

[7] A. Denning, *ActiveX Controls Inside Out*, Microsoft Press, 1997.

[8] G. Hamilton, *JavaBeans Specification 1.01*, Sun Microsystems, 1997.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[10] Object Management Group, *OMG Unified Modeling Language Guide Specification Version 1.4*, 1999, http://www.uml.org/, Last visited June 6, 2003.

[11] A.J.A. Wand, Reuse Metrics and Assessment in Component-Based Development, In *Proc. of 6th IASTED International Conference on Software Engineering and Applications*, IASTED, 2002, pp. 583–588.

[12] J.A. McCall, P.K. Richards and G.F. Walters, Factors in Software Quality, *US Rome Air Development Center Reports*, Vol. I, II, III, RADC-TR-77-369, 1977.

[13] ISO/IEC Standard ISO-9126, Software Product Evaluation-Quality Characteristics and Guidelines for Their Use, 1991.

[14] G. Sindre, R. Conradi and E. Karlsson, The REBOOT Approach to Software Reuse, *Journal of Systems and Software*, Vol.30, No.3, Elsevier, 1995, pp. 201–212.

[15] BeanBox, http://java.sun.com/products/javabeans/, Last visited June 6, 2003.

[16] FukaBeans, http://www.fuka.info.waseda.ac.jp/, Last visited June 6, 2003.

[17] J. Han, A Comprehensive Interface Definition Framework for Software Components, In *Proc. of 5th Asia-Pacific Software Engineering Conference*, IEEE CS, 1998, pp. 110–117.

[18] E.L. Lehmann, *Nonparametrics: Statistical Methods Based on Ranks*, Holden-Day, 1975.

[19] JARS.COM, http://www.jars.com/, Last visited June 6, 2003.

[20] H. Aman et al., A Quantitative Method of Verifying Metrics Using Principal Component Analysis and Correlation Analysis, *Journal of IEICE*, Vol.J85-D-I, No.10, IEICE, 2002, pp. 1000–1002 (in Japanese).

[21] G.W. Snedecor and W.G. Cochran, *Statistical Methods*, Iowa State University Press, 1980.

[22] L.H. Etzkorn et al., Automated reusability quality analysis of OO legacy software, *Information and Software Technology*, Vol.43, No.5, Elsevier, 2001, pp. 295–308.

[23] L.H. Rosenberg and L.E. Hyatt, Software Quality Metrics for Object-Oriented Environments, *The Journal of Defense Software Engineering*, STSC, April, 1997.

[24] N. Warren and P. Bishop, *Java in Practice: Design Styles and Idioms for Effective Java*, Addison-Wesley, 1999.

[25] E. Cho, M. Kim and S. Kim, Component Metrics to Measure Component Quality, In *Proc. of 8th Asia-Pacific Software Engineering Conference*, IEEE CS, 2001.