A Metrics Suite for Measuring Quality Characteristics of JavaBeans Components

Hironori Washizaki, Hiroki Hiraguchi, and Yoshiaki Fukazawa

Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan
{washi, h_hira, fukazawa}@fuka.info.waseda.ac.jp

Abstract. In component-based software development, it is necessary to measure the quality of components before they are built into the system in order to ensure the high quality of the entire system. However, in application development with component 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. Moreover, conventional techniques do not cover the whole of quality characteristics. In this paper, we propose a suite of metrics for measuring quality of JavaBeans components based on limited information that can be obtained from the outside of components without any source codes. Our suite consists of 21 metrics, which are associated with quality characteristics based on the ISO9126 quality model. Our suite utilizes the qualitative evaluation data available on WWW to empirically identify effective metrics, and to derive a reference value (threshold) for each metric. As a result of evaluation experiments, it is found our suite can be used to effectively identify black-box components with high quality. Moreover we confirmed that our suite can form a systematic framework for component quality metrics that includes conventional metrics and newly defined metrics.

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]. In this paper, we use object-oriented (OO) programming language for the implementation of components. CBD does not always have to be object-oriented; however, it has been indicated that using OO paradigm/language is a natural way to model and implement components[2].

Low-quality individual components will result in an overall software package of low quality. It is therefore important to have product metrics for measuring the quality of component units. A variety of product metrics have been proposed for components[3,4,5,6]; however, nobody has so far reported on the results of a comprehensive investigation of quality characteristics.

In this paper we propose a suite of metrics that provide a comprehensive mechanism for judging the quality characteristics of high-quality black-box components, chiefly from the viewpoint of users.

A. Jedlitschka and O. Salo (Eds.): PROFES 2008, LNCS 5089, pp. 45–60, 2008.
© Springer-Verlag Berlin Heidelberg 2008

2 Component-Based Development and JavaBeans

A component is a replaceable/reusable software unit that provides a certain function. Components are generally implemented in an object-oriented programming language. Component-based development is a method for determining the software architecture (component architecture) that forms a development platform, reusing executable components or developing new components according to the architecture standard, and combining the resulting components to develop new software.

With the appearance of comprehensive development environments based on a visual component assembly metaphor and the popularization of environments for implementing web applications (JSP, ASP, etc.), client components have already become popular way of implementing items such as GUI components and general-purpose logic components[8]. Therefore this paper is concerned with JavaBeans components[10] as the subject of quality measurements.

2.1 JavaBeans Technology

JavaBeans is a component architecture for developing and using local components in the Java language. A JavaBeans component ("bean") is defined as a single class in the Java language that satisfies the two conditions listed below. Accordingly, a bean has constructors, fields and methods, which are the constituent elements of ordinary classes.

- It has an externally accessible default constructor that does not take any arguments, and can be instantiated simply by specifying the class name.
- It includes a `java.io.Serializable` interface and is capable of being serialized.

Figure 1 shows the UML class diagram of an example of a bean. In this example, the `Chart` class is a bean according to this definition. In JavaBeans, in addition to the above mentioned definition, it is recommended that the target class and associated classes conform to the following mechanism to make it easier for them to be handled by development environments and other beans:

- Properties: A property is a named characteristic whose value can be set and/or got (read) from outside the bean. In target classes that are handled as beans, a property is defined by implementing a setting method that allows the value of a characteristic to be externally set, and a getting method that allows the value of a characteristic to be externally read. Methods of both types are called property access methods. Property access methods are chiefly implemented according to the naming rules and the method typing. When the target class has a `getXyz()` method that returns a value of type `A` (or a `setXyz()` method that requires a argument value of type `A`), then it can be inferred that the class has a writable (or readable) property `xyz`. Most of a bean's properties tend to have a one-to-one correspondence to the fields implemented in the bean class[5]). In the example shown in Fig 1, the `Chart` class has the methods `setTitle` and `getTitle()` for setting and

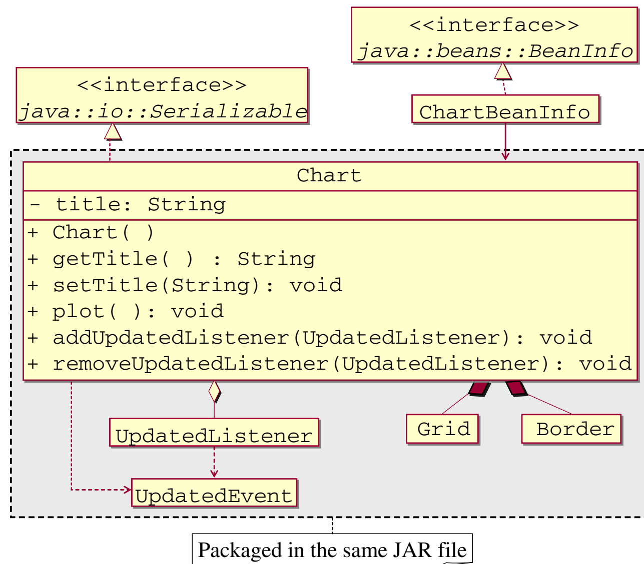


Fig. 1. Example of a bean and its associated classes (UML class diagram)

getting the value of a `title` field. Accordingly, the `Chart` bean has a `title` property whose value can be set and got.
- Methods: A method is a function that is provided for external interaction with a bean. In target classes that are handled as beans, they are defined by implementing `public` methods that can be called externally. In the example of Fig 1, the `Chart` bean has a `plot()` method.
- Events: An event is a mechanism for externally announcing that certain circumstances have arisen inside a bean. The constituent elements of an event are an event source, and event listener, and an event object. In the example of Fig. 1, the `Chart` bean has an `Updated` event.

The above mentioned definitions and mechanisms do not guarantee that it will exist in an environment where the other classes and/or interfaces on which the bean depends are present when the bean is independently distributed. Therefore, in JavaBeans it is recommended that a JAR archive file is used to store all the Java classes and interfaces on which the bean depends in the same archive file for distribution and reuse. In the example of Fig 1, the `Grid` and `Border` classes and event-related classes and interfaces on which the `Chart` bean depends must be distributed by storing them all together in a single JAR file.

2.2 JavaBeans Public Information

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[9]. Therefore, users who want to

reuse components often cannot obtain source codes of the components except for object codes. To allow a bean to be reused as a black-box component while keeping all its internal details hidden, the following information can be obtained externally without having to analyze the source code.

- Basic bean information: An introspection mechanism[10] can be used to obtain information about the properties, events and methods of the above mentioned mechanism. This information is obtained either based on the naming rules, or by analyzing a BeanInfo object provided by the bean developer.
- Class constituent information: Information relating to the constructors, fields and methods of the bean as a class can be obtained by using a reflection mechanism.
- Archive file structure information: Information about the structure of the archive file containing the bean (information on the presence or absence of various resource files such as icons, and the constituent elements of other classes/interfaces on which the bean depends) can also be obtained externally without analyzing the source code.

This externally accessible public information is an essential judgment resource for measuring the quality characteristics of a bean.

3 Component Quality Metrics

To evaluate a component when it is reused, the component is assessed from a variety of viewpoints (e.g., maintainability, reusability, and so on)[11]. This necessitates the use of metrics that consistently deal with the overall quality provided by a component rather than a single metric that deals with a single quality characteristic of a component.

Since beans are implemented in Java, it is possible to apply the quality measurements of conventional object-oriented product metrics. However, most conventional metrics perform measurements on entire object-oriented systems consisting of class sets. On the other hand, since components are highly independent entities, it is difficult for these metrics to reflect the component characteristics even when applied to individual component units.

Also, conventional metrics often require analysis of the target source code. Components are sometimes distributed to and reused by third parties across a network, and since in this case they are black-box components whose source code cannot be seen by the user, it is impossible to use conventional white-box metrics[5]. Accordingly, for components whose source code is not exposed, we need measurements that can be applied in a black-box fashion.

In this paper, based on these issues, we use the following procedure to construct a suite of metrics that provide a component's user with useful materials for making judgments when a component is reused.

1. Comprehensive investigation of basic metrics
2. Selection of basic metrics based on qualitative assessment information
3. Construction of a suite of metrics

3.1 Comprehensive Investigation of Basic Metrics

All the information that can be measured from outside a bean is comprehensively investigated as basic metrics. The investigation results are shown in Table 1. Tables 1(a), (b) and (c) show the metrics relating to the bean's information, class structure information, and archive file structure information resepectively.

In Table 1(a), "Default event present" expresses whether an initially selected event is pre-specified when a bean that provides multiple events is used in a development environment. Similarly, "Default property present" expresses whether an initially selected property is pre-specified.

In Table 1(b), RCO and RCS are the ratios of property getting methods and setting methods out of all the bean fields, and are used as metrics expressing the extent to which the properties of fields can be publicly got and set[5]. SCCp and SCCr are the ratios of methods that have no arguments or return values, and are used as metrics expressing the independence of the methods[5]. PAD and PAP are the ratios of `public`/`protected` methods and fields, and are used as metrics expressing the degree to which the methods and fields are encapsulated[12].

In Table 1(c), the notation "Overall M" represents the results of applying metric M under conditions where the constituent elements of all the classes contained in the archive file that includes the bean are assumed to exist within a single class. The number of root classes expresses the number of classes that are direct descendents of `java.lang.Object`. Also, "Overall bean field (method) ratio" expresses the ratio of fields and methods that a bean has in the sum total of fields (methods) in the entire classes.

3.2 Selection of Basic Metrics

Out of all the resulting basic metrics, we select those that are useful for judging the level of quality of the component. For this selection we use manually obtained component evaluation information published at jars.com[13]. The evaluation information at jars.com has already been used to set the evaluation standard values of a number of metrics[5,6]. At jars.com, in-house or independent group of Java capable and experienced individuals review each bean from the viewpoints of presentation, functionality and originality. Finally beans are rated into 8 levels as total of those different viewpoints. These 8 evaluation levels are normalized to the interval [0, 1] (where 1 is best), and the resulting value is defined as the JARS score.

As our evaluation sample, we used all of the 164 beans that had been evaluated at jars.com as of March 2004. The publication of beans at jars.com means that they are reused in unspecified large numbers, so the JARS score is thought to reflect the height of the overall quality of the component taking the fact that the bean is reused into account. We therefore verified the correlation between the measured values of each bean's basic metrics and its JARS score.

As the verification method, we divided the components into a group with a JARS score of 1 (group A: 117 components) and a group with a JARS score of less than 1 (group B: 47 components), and we applied the basic metrics to all the beans belonging to each group. In cases where testing revealed a difference between the measured value distributions of each group, this basic metric

Table 1. Possible basic metrics relating to: (A: normality test result of "good" components group, B: that of "poor" group, T: difference test result of both distributions)

(a) bean itself

Metric M	A	B	T
`BeanInfo` present	n	n	n
Number of events	n	n	Y
Number of methods (of bean)	n	n	Y
Number of properties	n	n	n
Default event present	n	n	n
Default property present	n	n	n

(b) class constituent

Metric M	A	B	T
Number of fields	n	n	Y
RCO	n	n	Y
RCS	n	n	Y
`abstract` field ratio	n	n	n
`final` field ratio	n	n	n
`private` field ratio	n	n	n
`protected` field ratio	n	n	Y
`public` field ratio	n	n	n
`static` field ratio	n	Y	n
`transient` field ratio	n	n	n
`volatile` field ratio	n	n	n
PAD	n	n	Y
Number of constructors	n	n	Y
Constructor without arguments (default constructor) ratio	n	n	Y
Average number of arguments per constructor	n	n	Y
`private` constructor ratio	n	n	n
`protected` constructor ratio	n	n	n
`public` constructor ratio	n	n	n
Total number of methods	n	n	n
SCCp	Y	Y	Y
SCCr	n	n	n
Average number of arguments per method	Y	n	n
`abstract` method ratio	n	n	n
`final` method ratio	n	n	n
`native` method ratio	n	n	n
`private` method ratio	n	n	n
`protected` method ratio	n	n	Y
`public` method ratio	n	Y	n
`static` method ratio	n	n	Y
`strictfp` method ratio	n	n	n
`synchronized` method ratio	n	n	Y
PAP	n	Y	n

(c) archive file constituent

Metric M	A	B	T
Number of files	n	n	Y
Class file ratio	n	Y	n
Number of icons	n	n	Y
Number of classes	n	n	Y
Number of root classes	n	n	n
Average depth of class hierarchy (DIT)	Y	Y	Y
Abstract class ratio	Y	n	n
`final` class ratio	n	n	Y
Interface ratio	n	n	n
`private` class ratio	n	n	Y
`protected` class ratio	n	n	Y
`public` class ratio	n	n	n
`static` member class ratio	n	n	n
`synchronized` class ratio	n	n	n
Overall number of fields	n	n	Y
Average number of fields per class	n	n	n
Overall RCO	n	n	Y
Overall RCS	n	n	Y
Overall `abstract` field ratio	n	n	n
Overall `final` field ratio	n	Y	n
Overall `private` field ratio	Y	Y	n
Overall `protected` field ratio	n	n	n
Overall `public` field ratio	n	Y	n
Overall `static` field ratio	n	Y	n
Overall `transient` field ratio	n	n	n
Overall `volatile` field ratio	n	n	n
Overall PAD	n	n	n
Overall number of constructors	n	n	Y
Average number of constructors per class	n	Y	n
Overall constructor without arguments (default constructor) ratio	n	Y	n
Overall average number of arguments per constructor	Y	Y	n
Overall `private` constructor ratio	n	n	n
Overall `protected` constructor ratio	n	n	Y
Overall `public` constructor ratio	n	n	n
Overall number of methods	n	Y	n
Average number of methods per class	n	n	n
Overall SCCp	n	n	Y
Overall SCCr	n	n	n
Overall average number of arguments per method	n	Y	n
Overall `abstract` method ratio	n	n	n
Overall `final` method ratio	n	n	n
Overall `native` method ratio	n	n	n
Overall `private` method ratio	Y	n	n
Overall `protected` method ratio	n	Y	n
Overall `public` method ratio	Y	Y	n
Overall `static` method ratio	n	Y	n
Overall `strictfp` method ratio	n	n	n
Overall `synchronized` method ratio	n	n	n
Overall PAP	Y	Y	n
Overall bean field ratio	n	n	Y
Overall bean method ratio	n	n	Y

was judged to affect the JARS score and was selected as a metric constituting the suite of metrics. Tests were performed for each metric M according to the following procedure.

1. With regard to the distribution of the measured value of M in each group, we tested for normality at a critical probability of 5%. The test results of group A and group B are respectively shown in columns A and B of Table 1. `Y` indicates that the results were normal, and `n` indicates that the results were not normal.
2. We tested the differences in the distributions of the measured values in both groups. When both groups were found to be normal, we used Welch's t-test[14] to check whether or not both groups had the same population mean. In other cases, we used the Mann-Whitney U-test[14] to check whether or not the median values of both population distributions were the same. These test results are shown in the `T` column of Table 1. `Y` indicates that the distributions were found to be different; i.e. there is a possibility to classify each bean into two groups by using the target metric.

3.3 Construction of Quality Metrics Suite

As a result of these tests, we found differences in the distributions of the measured values between the two groups for 29 metrics. Below, we will consider the association of these measurement test results with quality characteristics in the ISO9126 quality model[7][1].
· Number of events: Figure 2(a) shows a box-and-whisker plot of the measurement results. This box-and-whisker plot shows the range of the measured values together with the 25%/75% quantiles and the median value for group A (JARS score = 1; left side) and group B (JARS score < 1; right side). The measured values tended to be higher in group A. It seems that beans with a large number of events have sufficient necessary functions and a higher level of suitability.
· Number of methods: According to Fig. 3(a), there tended to be more methods in group A. Beans with a greater number of methods are able to fully perform the required operations and have a greater level of suitability.
· Number of fields: According to Fig. 3(b), the number of fields tends to be smaller in group A. This is thought to be because when using a bean in which the number of fields has been suppressed, the user is unaware of the unnecessary fields, resulting n greater understandability.
· Ratio of `protected` fields: No differences were observed in the distributions of measured values relating to fields with other types of visibility (private/public), so it appears that field visibility does not affect a bean's quality. This metric was therefore excluded from the suite.
· Ratio of `protected` methods: No differences were observed in the distributions of measured values relating to methods with other types of visibility (private/public), so it appears that method visibility does not affect a bean's quality. This metric was therefore excluded from the suite.

[1] Although several problems such as ambiguity have been indicated for the ISO9126 model[15] it can be a good starting point to explore related quality characteristics.

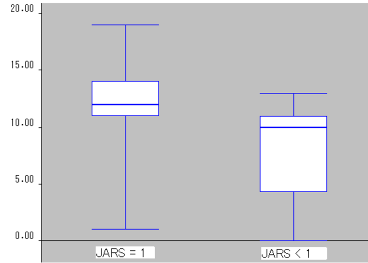
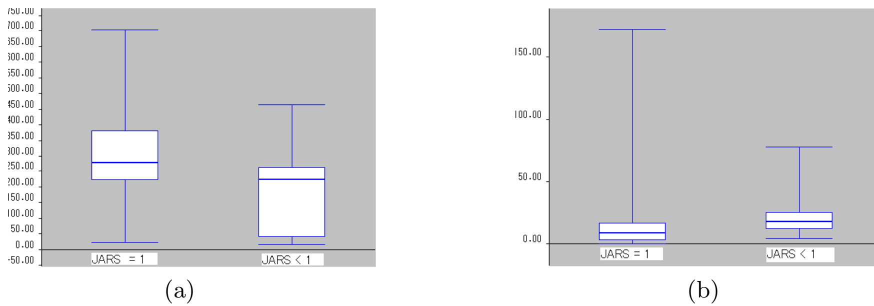


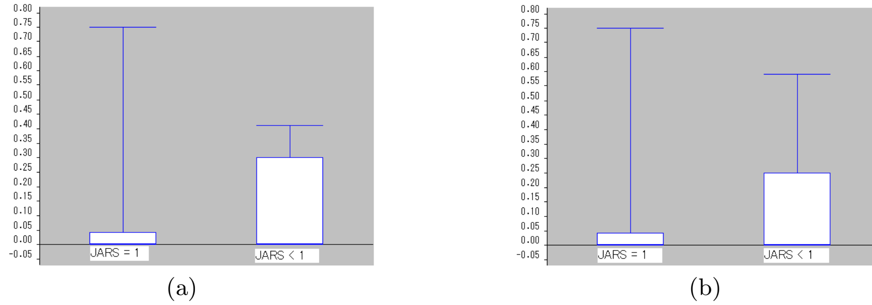
Fig. 2. Number of events (p-value of the null hypothesis=0.0007)



(a) (b)

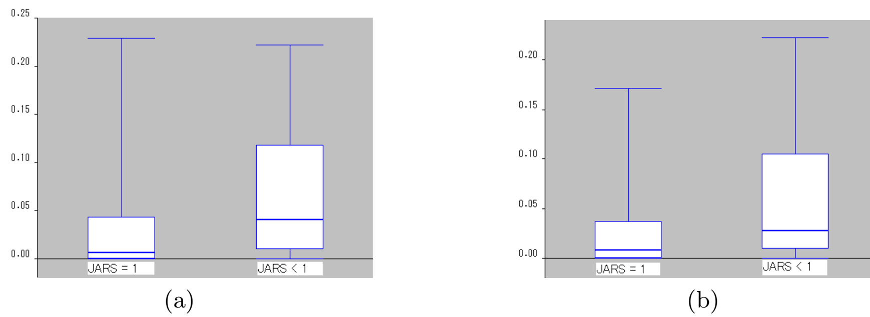
Fig. 3. (a) Number of methods (0.0068) (a) Number of fields (0.0008)

· RCO and RCS: According to Fig. 4(a) and (b), both of these measured values tended to be smaller in group A. By suppressing the number of properties that can be got/set, it is possible to reduce access as properties to more fields than are necessary, which is thought to result in a high level of maturity. Also, since the user is not bothered with unnecessary fields when using the bean, it is though that the understandability is high.
· Overall RCO and overall RCS: According to Fig. 5(a) and (b), both of these measured values tended to be smaller in group A. Unlike the bean RCO/RCS values, the overall RCO/RCS values are thought to represent the internal maturity and stability of a bean.
· PAD: According to Fig. 6(a), this measured value tended to be smaller in group A. In a bean where this measured value is small, there are few fields that can be operated on without using property access methods, so it is thought that the maturity and changeability are high.
· Number of constructors: According to Fig. 6(b), this measured value tended to be larger in group A. When there is a large number of constructors, it is possible to select a suitable constructor when the class is instantiated, so it is thought that the suitability and testability are high.
· Default constructor ratio: This measured value tended to be smaller in group A. However, since all beans must by definition have a default constructor, this



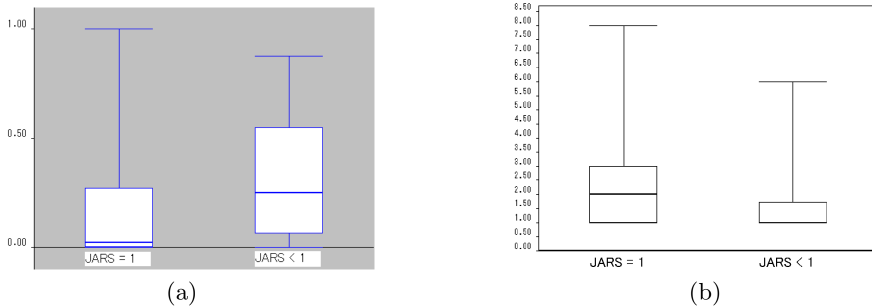
(a) (b)

Fig. 4. (a) RCO (0.0671) (b) RCS (0.1096)



(a) (b)

Fig. 5. (a) Overall RCO (0.0061) (b) Overall RCS (0.0071)



(a) (b)

Fig. 6. (a) PAD (0.0042) (b) Number of constructors (0.0031)

metric exhibited the same tendency as the number of constructors. Accordingly, this metric is redundant and is excluded from the suite.

· Average number of arguments per constructor: This measured value tended to be larger in group A. However, since all beans must by definition have a void constructor, this metric exhibited the same tendency as the number of constructors. Accordingly, this metric is excluded from the suite.

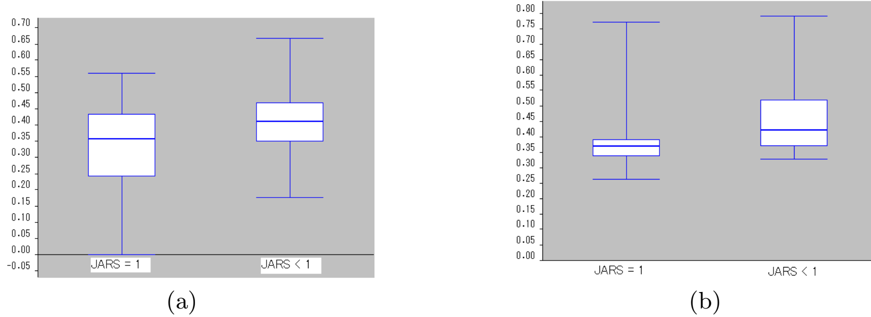


Fig. 7. (a) SCCp (0.0049) (b) Overall SCCp (0.0004)

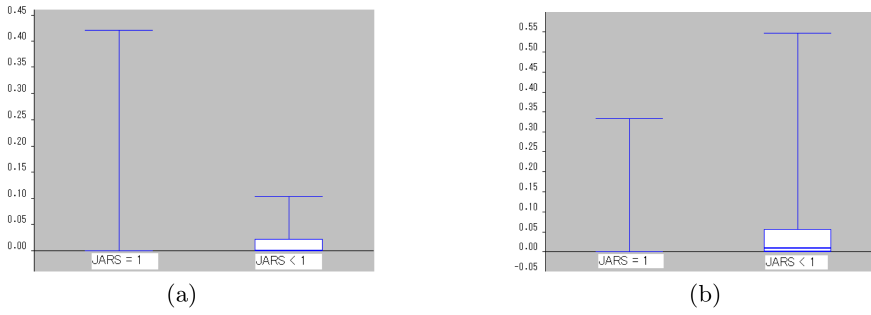
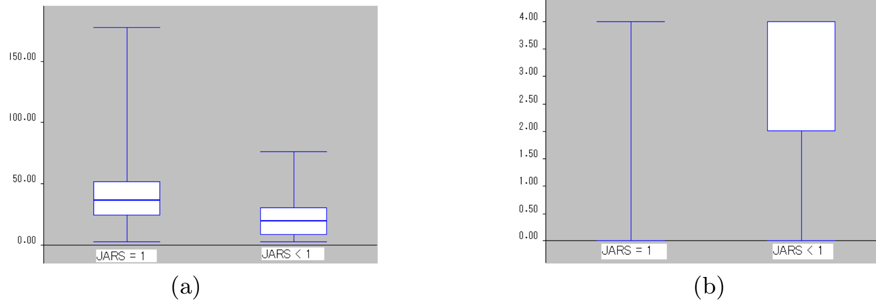


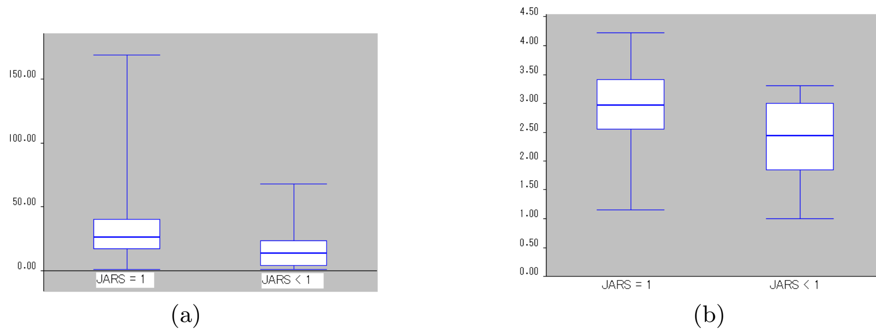
Fig. 8. (a) Static method ratio (0.0857) (b) Synchronized method ratio (0.0083)

· SCCp and overall SCCp: According to Fig. 7(a) and (b), the measured values for both of these metrics tended to be smaller in group A (where there is a higher proportion of methods with no arguments). It is thought that the understandability and analyzability are high because less information has to be prepared at the user side when the methods are used. Here, the overall SCCp differs from the SCCp of individual beans in that there is no redundancy because it relates to the handling of methods inside the bean.
· `static` method ratio: According to Fig. 8(a), the measured values tended to be smaller in group A. When there are few static methods, the possibility of being operated from various locations without instantiating a bean is reduced, so it is thought that that the analyzability is high.
· Synchronized method ratio: According to Fig. 8(b), this measured value tended to be smaller in group A. When there are few synchronized methods, it is thought that the target bean is set up so that it can be used either in multi-thread or single-thread environments, thus resulting in high analyzability.
· Number of files: According to Fig. 9(a), this measured value tended to be larger in group A. However, since the measured value of the number of files is more or less proportionally related to the number of classes, it is thought that the number



(a) (b)

Fig. 9. (a) Number of files (0.0005) (b) Number of icons (0.0641)



(a) (b)

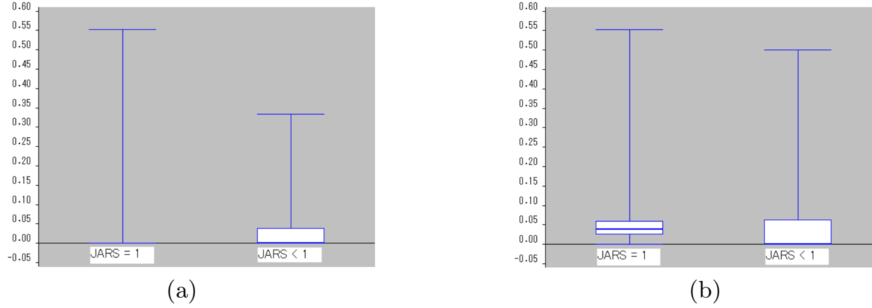
Fig. 10. (a) Number of classes (0.0009) (b) Average depth of class hierarchy (0.0009)

of classes is a more suitable indicator of the scale of a bean. Accordingly, the number of files is excluded from the suite.

· Number of icons: According to Fig. 9(b), this measured value tended to be larger in group A. Icons are information used to represent beans when they are selected in the development environment, and the magnitude of this measured value is thought to reflect the degree of operability.

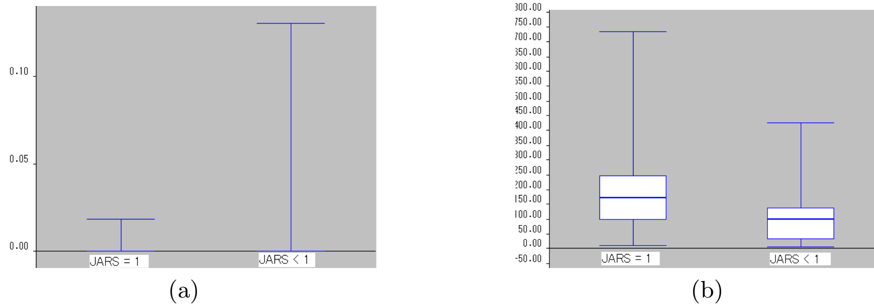
· Number of classes: According to Fig. 10(a), the number of classes tended to be larger in group A. Looking at the results for other metrics, there is no difference between the distributions of group A and group B in terms of the average number of fields per class and the average number of methods per class, so it is thought that beans with a large number of classes in the archive are not dependent on class sets that are fragmented any more than is necessary, but that they purely express more functions. Therefore, it is thought that beans with more classes have higher suitability.

· Average depth of class hierarchy (DIT[16]): According to Fig. 10(b), this measured value tended to be larger in group A. In object-oriented design, the reuse of fields/methods and the embodiment of variable parts are realized by differential definitions based on inheritance. Therefore, it is thought that the analyzability



(a) (b)

Fig. 11. (a) Final class ratio (0.0585) (b) Private class ratio (0.0074)



(a) (b)

Fig. 12. (a) Protected class ratio (0.2509) (b) Overall number of fields (0.0016)

and changeability both increase with increasing depth of the inheritance hierarchy in the class set that constitutes the archive.

· `final`/`private`/`protected` class ratio: According to Fig. 11(a) and (b) and Fig. 12(a), all three of these metrics tended to be smaller in group A. Since these measured values are not encapsulated in a bean any more than is necessary, it is thought that the testability is high.

· Overall number of fields: According to Fig. 12(b), this measured value tended to be larger in group A. Since there is no difference between the two groups in terms of the distribution of the average number of fields per class, it is thought that this measured value increases as the number of classes increases, regardless of how high the quality is. Therefore, this metric is excluded from the suite because it represents the same characteristic as the number of classes.

· Overall number of constructors: This measured value tended to be larger in group A. Since there was no difference between the two groups in terms of the distribution of the average number of constructors per class, it is thought that this measured value increases as the number of classes increases, regardless of how high the quality is. Therefore, this metric is excluded from the suite because it represents the same characteristic as the number of classes.

· Overall `protected` constructor ratio: No differences were observed in the distributions of measured values relating to constructors with other types of visibility (private/public), so it appears that the visibility of constructors in the class set

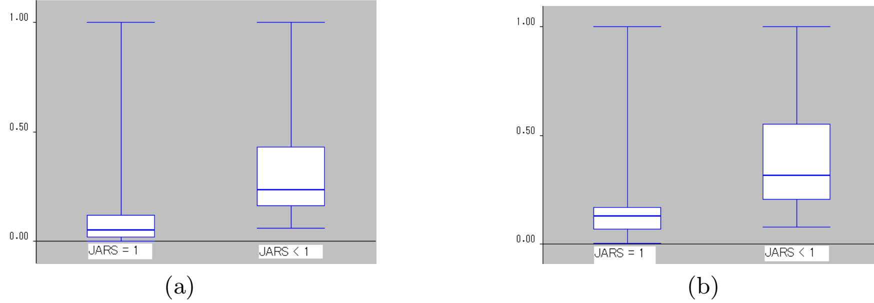


Fig. 13. (a) Overall bean field ratio (0.0000) (b) Overall bean method ratio (0.0000)

constituting an archive does not affect a bean's quality. This metric is therefore excluded from the suite.
· Overall bean field ratio/bean method ratio: According to Fig. 13(a) and (b), this measured value tended to be smaller in group A for both of these metrics. In beans where these measured values are small, the realization of required functions is transferred to (fields/method in) other dependent class sets while suppressing information that is published externally, so it is thought that the maturity and analyzability are high.

Based on these findings, we selected 21 metrics to be incorporated in the quality metrics suite. According to our consideration of the results, Figure 14 shows a framework for component quality metrics (i.e. the suite of quality metrics) in which these metrics are associated with the quality characteristics mentioned in the ISO9126 quality model. In Fig. 14, metrics that are thought to be effective for measuring the quality of beans are connected by lines to the quality subcharacteristics that are closely related to these metrics in order to show their linked relationships. Of the 21 metrics, 14 metrics obtained results relating to maintainability.

Using this framework, it is possible to make detailed quality measurements focused on beans, and to select metrics that efficiently and comprehensively take account of the overall bean quality.

4 Verifying the Validity of the Metrics Suite

4.1 Standard Assessment Criteria

As a threshold value for deciding whether a target bean belongs in either group A or group B of the previous section, we obtained standard assessment criteria for each basic metric. Using these standard assessment criteria, we verified whether or not it is possible to judge the quality of a bean. If $\overline{X_{M,a}}$ and $\overline{X_{M,b}}$ are the average values of metric M in group A and group B respectively, then the standard assessment criterion E_M is defined as follows:

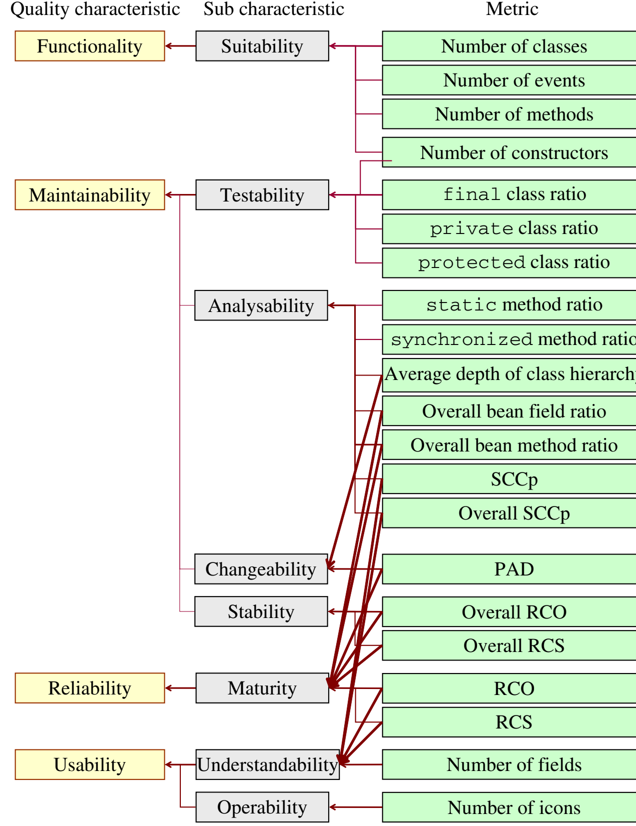


Fig. 14. A framework of component quality metrics

$$E_M = \begin{cases} \text{More than or equals to } \frac{\overline{X_{M,a}}+\overline{X_{M,b}}}{2} & (\text{if } \overline{X_{M,a}} > \overline{X_{M,b}}) \\ \text{Less than or equals to } \frac{\overline{X_{M,a}}+\overline{X_{M,b}}}{2} & (\text{Otherwise}) \end{cases}$$

When a measured value corresponds to a standard assessment criterion, the corresponding quality characteristics and/or sub-characteristics of the bean are high. For each of the 21 metrics constituting the proposed metrics suite, Table 2 lists the proportion of beans in group A that correspond to the standard assessment criterion (conformity R_A) and the proportion of beans in group B that do NOT correspond to the standard assessment criterion (conformity R_B). If both of R_A and R_B are close to 100%, the target standard assessment criterion is almost perfectly useful to classify each bean into two groups.

As both degrees of conformity become higher, it shows that the metric is more effective at correctly measuring the quality of the target bean and classifying it into the correct group. According to Table 2, the conformity values are both 50% or more for nine metrics such as SCCp, which shows that these nine metrics are particularly effective at quality measurements. Also, since the overall average

Table 2. Standard assessment criteria and conformity

Metric M	E_M	R_A	R_B
Number of events	≥ 11	77%	57%
Number of methods	≥ 248	60%	61%
Number of icons	≥ 4	99%	26%
Number of classes	≥ 31	45%	83%
Average depth of class hierarchy	≥ 2.6	69%	57%
Final class ratio	$\leq 3\%$	96%	26%
Private class ratio	$\leq 6\%$	79%	30%
Protected class ratio	$\leq 0.8\%$	98%	17%
Overall RCO	$\leq 4\%$	78%	48%
Overall RCS	$\leq 5\%$	78%	39%
Overall SCCp	$\leq 42\%$	91%	52%
Number of fields	≤ 18	76%	48%
RCO	$\leq 9\%$	80%	39%
RCS	$\leq 9\%$	76%	43%
PAD	$\leq 25\%$	74%	52%
Number of constructors	$\geq 2\%$	63%	78%
SCCp	$\geq 66\%$	51%	61%
Static method ratio	$\leq 1.5\%$	84%	35%
Synchronized method ratio	$\leq 4\%$	93%	35%
Overall bean field ratio	$\leq 22\%$	91%	57%
Overall bean method ratio	$\leq 27\%$	87%	52%
Average	–	74%	50%

values for both types of conformity are equal to or over 50%, it is highly likely that the quality of a bean can be suitably assessed by using the combination of multiple metrics constituting the proposed metrics suite.

4.2 Comparison with Conventional Metrics

Metrics suitable for beans in situations where the source code is unavailable include the usability metrics of Hirayama et al.[6], the testability metrics summarized by R. Binder[12], and the object-oriented metrics of Chidamber and Kemerer[16]. Of these conventional metrics, our proposed metrics suite includes all the metrics that can be applied to beans. The contribution of this paper is that it proposes a systematic framework for component quality metrics that includes these existing metrics and newly defined metrics, and that it has been verified using qualitative assessment information.

Metrics for measuring the complexity and reusability of beans have been proposed by Cho et al.[3], but these metrics included the need for analysis of the bean source code. Wang also proposes metrics for measuring the reusability of JavaBeans components[4]; however the metrics indicate the actual reuse rates of the reused component in a component library and cannot be used in a situation where sufficient time has not passed since the target component was developed.

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

5 Conclusion and Future Work

We have proposed metrics for evaluating the overall quality of individual JavaBeans components in a black-box fashion, and we have empirically confirmed that they are effective based on a correlation with the resulting qualitative assessment information.

In the future, by carrying out manual verification trials, we plan to make a detailed verification of the effectiveness of these proposed metrics, and of the validity of the association between each metric and the quality characteristics. Several metrics that constitute the proposed metrics suite can also be applied to ordinary Java classes that are not beans. We also plan to investigate the possibility of applying them to other classes besides beans.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1999)
2. Hopkins, J.: Component Primer. Communications of the ACM 43(10) (2000)
3. Cho, E., Kim, M., Kim, S.: Component Metrics to Measure Component Quality. In: Proc. 8th Asia-Pacific Software Engineering Conference (2001)
4. Wand, A.J.A.: Reuse Metrics and Assessment in Component-Based Development. In: Proc. 6th IASTED International Conference on Software Engineering and Applications (2002)
5. Washizaki, H., et al.: A Metrics Suite for Measuring Reusability of Software Components. In: Proc. 9th IEEE International Symposium on Software Metrics (2003)
6. Hirayama, M., Sato, M.: Usability evaluation of software components. IPSJ Journal 45(6) (2004)
7. ISO/IEC 9126 International Standard: Quality Characteristics and Guidelines for Their Use (1991)
8. Suzuki, M., Maruyama, K., Aoki, T., Washizaki, H., Aoyama, M.: A Research Study on Realization of Componentware Technology, Research Institute of Software Engineering (2003)
9. Aoyama, M., et al.: Software Commerce Broker over the Internet. In: Proc. 22nd IEEE Annual International Computer Software and Applications Conference (1998)
10. Hamilton, G.: JavaBeans 1.01 Specification, Sun Microsystems (1997)
11. Sedigh-Ali, S., et al.: Software Engineering Metrics for COTS-Based Systems, Computer, vol. 34(5) (2001)
12. Binder, R.: Design for Testability in Object-Oriented Systems. Communications of the ACM 37(9) (1994)
13. JARS.COM: Java Applet Rating Service, `http://www.jars.com/`
14. Glass, G.V., Hopkins, K.D.: Statistical Methods in Education and Psychology. Allyn & Bacon, MA (1996)
15. Al-Kilidar, H., et al.: The use and usefulness of the ISO/IEC 9126 quality standard. In: Proc. 4th International Symposium on Empirical Software Engineering (2005)
16. Chidamber, S., Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering 20(6) (1994)