Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp912-918)

Visualization of Binary Component-Based Program Structure with Component Functional Size

Hironori Washizaki† Satoru Takano‡ Yoshiaki Fukazawa‡
†Research Center for Testbeds and Prototyping,
National Institute of Informatics
Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo, 101-8430
Japan
‡Dept. Computer Science, School of Science and Engineering,
Waseda University
Ohkubo 3-4-1, Shinjuku-ku, Tokyo, 169-8555
Japan
† washizaki@nii.ac.jp http://www.fuka.info.waseda.ac.jp/

Abstract: In this paper, we propose a program visualization system which does not make use of the source code, but uses two techniques, reflection and byte-code analysis, to measure the functional size of each software component and to determine the dependency relationships among components and helper classes. These results are used to provide an accurate visualization of the overall structure of the component-based program. Our system can be applied to programs built with JavaBeans components. As a result of comparative evaluations, it is found that our system is useful for visualizing binary component-based program structure with component functional size to support maintenance activities.

Key-Words: Program visualization, Component-based development, Program comprehension, Software reuse, Object-oriented programming, JavaBeans

1 Introduction

Component-based development (CBD) [1,2,3] is an approach allowing development of more-versatile, large-scale software more quickly and effectively. Often in CBD, components that have been developed by a third party and delivered in binary format (without access to the source code) are reused to build new software quickly. These types of components will be called *binary components* in the remainder of this paper.

It is well known that much of the time spent maintaining software is consumed in simply understanding the software [4]. In order to effectively maintain software that has been obtained through CBD on an on-going basis, it is necessary to provide the maintainer with an intuitive understanding of the software as a collection of components. This is usually done by expressing various facets of the large amount of information graphically, through visualization.

Two important aspects of software are its static structure and its dynamic behavior. This paper will deal mainly with the former, discussing visualization of the static structure of component-based program. With most of the existing systems and techniques for visualizing software structure (e.g. SEIV[4], Seesoft[5], ObjectOrrery[6], etc.), analysis of the program source code is required, so using them to visualize software that was created by incorporating binary components is very difficult.

In this paper we propose a system which uses two techniques, reflection and byte-code analysis, to obtain the dependency relationships among components (and helper classes) without using source code, in programs composed of JavaBeans [7] binary components. Our system then visually displays a graphical representation of these relationships. It also applies functional size measurement values obtained using a new component-size metric developed by us.

2 CBD and JavaBeans

Components are independent, interchangeable and reusable units of software that provide a particular function [2]. They are generally implemented in an object-oriented programming language [8,9]. CBD refers to the process of developing new software by selecting a software architecture as a development platform (the component architecture), and combining reused and newly developed components that are compatible with the standards of the selected architecture.

In this paper we handle visualization of Java programs built by combining JavaBeans components [7]. JavaBeans is a component architecture for developing and using local components in the Java language. JavaBeans components are called *Beans*, and are defined by a single class in the Java programming language and must satisfy the two conditions below.

- Condition 1: The class must have a public constructor which requires no parameters.
- Condition 2: The class must implement the `java.io.Serializable` interface.

As such, a Bean has the structural constructs of a regular Java class: constructors, fields and methods. As an example, the static structure of a typical Bean is shown as a UML class diagram in Fig.1. In the example, the `Chart` class is a Bean.

In addition to the above definitions, it is recommended that Bean and related classes follow the conventions described below, which make the functions easier to be used by other Beans and easier to use in development environments.

- Properties: Named attributes whose value can be obtained and/or set externally are called *Properties*. For classes which are handled as Beans, properties are defined by implementing a method which allows the attribute value to be set externally, and another which allows the attribute value to be obtained externally. These are called the *property access methods*. Property access methods are usually implemented according to specific naming conventions. If a class has a method, `public A getXyz()`, and/or a method, `public void setXyz(A a)`, it is considered to have a property, `xyz`. For most of the properties of a Bean, there is a 1-to-1 correspondence between properties and fields of the Bean class [10]. In Fig.1, the `Chart` class has `setTitle()` and `getTitle()` methods which set and get the value of the title field. Thus, the `Chart` Bean has a `title` property whose value can be set and obtained.
- Methods: Functions provided by the Bean that can be called externally are called *methods*. Methods are implemented as externally callable (public) normal-Java methods in the Bean class. The `Chart` Bean in Fig.1 has a `plot()` method.
- Events: *Events* provide a mechanism for a Bean to notify external entities of particular occurrences internal to the Bean, when they happen. An Event consists of the event source, the event listener, and the event object. The example in Fig.1 is a Bean with an `Updated` event.

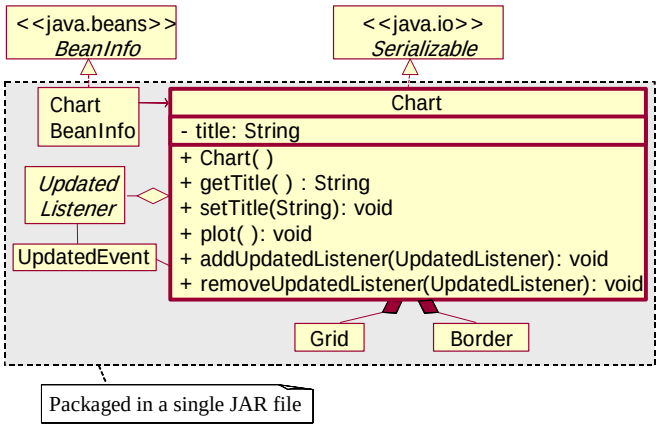

Fig.1: Example of a Bean and related classes

Moreover, it is recommended that all of the classes and interfaces that a Bean requires be packaged and delivered with the Bean in the same JAR archive file. The `Chart` Bean in the example in Fig.1 is distributed as a single JAR file which also contains the `Grid`, `Border` and classes/interfaces related to the `Updated` event.

For Beans supplied in binary format (Java byte-code), the following information 1-3 can be obtained from the Bean through the reflection (or introspection) mechanism without analyzing the source code. Moreover, the information in item 4 can be obtained from the Java byte-code by the byte code analysis [11].

1. Information about properties, events, and methods can be obtained through the introspection mechanism [7]. This information can be analyzed using the naming conventions, or by using a `BeanInfo` object containing meta-data if the developer has provided one.
2. Information about the constructors, fields and methods of the Bean class can be obtained using the reflection mechanism.
3. Structural information from the archive file which contains the Bean can also be obtained without analyzing the source code (e.g.: whether there are resource files such as icon

files, what other classes or interfaces the Bean depends upon, etc.).

4. Information about how the Bean's Java class uses or is used by other Beans or helper classes, how it generates instances of other classes, or how instances of it are generated by other Beans or classes, can be obtained by analyzing the byte-code.

This available information in 1-4 is important for determining the functional size of a Bean.

3 Program Visualization

The activity of visually displaying the structure or behavior of the final text of a software program, for the purpose of supporting software development is called program visualization. Program visualization can be categorized into four types according what is being visualized (the program or the data handled by the program), and the type of drawing technique used (static or dynamic) [12]. Among these types, this paper deals with a system for statically visualizing the component-based program.

Seesoft [5] and ObjectOrrery [6] are examples of the many existing types of static program visualization systems for visualizing object-oriented programs. For example, ObjectOrrery, for programs written in the Smalltalk object-oriented programming language, has a function which gathers the reference relationships among groups of classes and selects groups of them for display, and another which displays the scope and effect of particular changes to the class structure.

However, these conventional program visualization systems all require analysis of the source code of the program being visualized, so it has been very difficult to apply them to binary component-based programs built using components provided in binary format. Furthermore, these systems generally are not able to display particular characteristics (e.g. functional size) of individual components which make up the program within the component architecture being used.

4 Binary Component-Based Program Visualization

Programs created by combining helper classes and binary components without source code are referred to as *binary component-based program*. In order to help programmers gain the understanding of the software required to perform maintenance tasks such as fixing bugs or adding extensions efficiently, we propose a system which visually displays the static structure of a Java program made up of Java helper classes and JavaBeans components (Beans) provided in byte-code format.

Our system displays the functional size of each Bean, as well as the dependency relationships between Beans and other classes by using the Java reflection and JavaBeans introspection functions and by analyzing the byte-code of the components.

4.1 System architecture

Fig.2 shows our system architecture. Below, we show the data states in our system based on a visualization reference model [13].

- Raw data: Collection of Java byte-code
- Data table: Class/component dependency relationships and functional size of each component derived from the byte-code.
- Visualization structure: Dependency relationships and functional size information arranged within a 3-D space.
- Displayed data: Dependency relationships and functional size arranged on a 2-D surface so they can be shown on the display of a computing device.

Below, we show the operations in our system based on the visualization reference model [13].

- Data transform: The data table is created from the raw data by two parts: the component analysis and the dependency analysis. The former first determines whether a Java class satisfies the JavaBeans specifications, and if it is a Bean, its functional size is determined using the reflection mechanism. The latter uses the Javassist byte-code analysis tool [11] to analyze the Java byte-code, and obtains the dependency relationships between classes.
- Visualization mapping: The classes and components which comprise the raw data, and the dependency relationships between them are the visualization objects (units of visualization) and are represented within a 3-D coordinate space using "Jun for Java" [14], a 3-D graphics/multi-media framework. The visualization objects for the classes and components each have their own features (name, functional size, etc.).
- View transform: The 3-D coordinate data obtained using the visual mapping is output to the 2-D image using "Jun for Java".

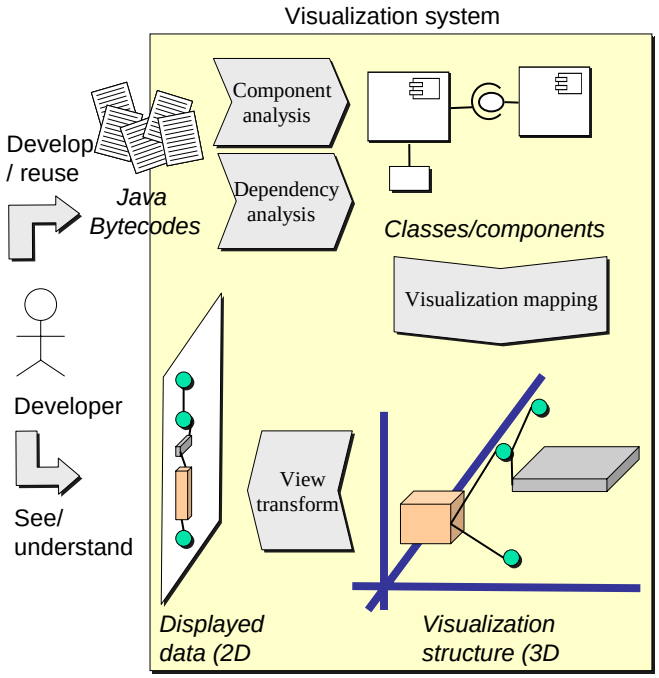

Fig.2: Overview of visualization system

4.2 Dependency analysis

The following dependency relationships between classes are obtained through analysis of the byte-code in the dependency analysis part.

- Inheritance relationships: When a Java class is defined as a subclass of another class by using the Java keyword `extends`, it is called an *inheritance relationship*.
- Reference relationships: When a class uses another class type or object, such as by using an object as a value, or by accessing a method or field of an object, it is called a *reference relationship*.
- Instance generation relationship: When a class or an object of the class internally generates an object of another class (using Java keyword `new`), it is called an *instance generation relationship*.

4.3 Component analysis

As described in section 2, component analysis consists of distinguishing between Java classes and Beans, and if the class is a Bean, using the reflection mechanism to determine the functional size of the Bean. The functional size of a component gives an indication of the amount of functionality provided by the component. By visually indicating the amount of functionality of each component that makes up a component-based program, our system can help give an intuitive understanding of the overall functional size, as well as whether the break-down and allocation of functionality within the program is appropriate.

The number of methods, properties and events made public by a component can be considered to be related to the functional size of that component.

- Number of methods: The number of methods a component has mainly reflects the extent of the provided functionality of the component that affects the component's internal state.
- Number of events: The number of events a component has mainly reflects the extent of the provided functionality of the component that affects component's external entities (i.e. other components and classes).
- Number of properties: Generally, when a method provided by a component is executed, it uses the values in specific properties, and places the results of this execution into specific properties [10]. Also, the events provided by the component are used to initiate external actions and notify external entities based on differing values in specific properties. Thus, the number of properties a component has reflects the input and output related to functionality provided internally, and the scope of the call-conditions for functionality provided externally.

The component analysis part measures these three values using the JavaBeans introspection function. However, as described above, each of these values reflects the functional size of the component from a different perspective. So, by combining these values, we define *FOC(c)*, the Functional size Of a Component, *c*, as a metric which comprehensively reflects the functional size. In the definition of FOC, we added one to each value before performing multiplication in order to avoid the final value of FOC become zero when one of these values is zero.

$$FOC(c) ::= (NOM(c)+1) \cdot (NOE(c)+1) \cdot (NOP(c)+1)$$

where *NOM(c)*, *NOE(c)*, and *NOP(c)* represent the number of methods, events and properties of *c*.

The value of this FOC metric of component functional size cannot be interpreted as indicating the component is better (or worse), the larger the value. For binary components whose internals are hidden, as the functional size increases, the component's applicability for reuse increases, but the effort required understanding the functionality

also increases, so it may also indicate additional problems in terms of maintenance.

This leads to the question of what is an appropriate value for the functional size of a binary component used in building component-based program. To compute a reference value for the proposed FOC metric, we used evaluation data from the contributed components made available on JARS.COM [15]. On JARS.COM, a large number of Beans in various categories are judged (by development experts) and given an 8-level evaluation with respect to expressiveness, functionality and originality. This 8-level evaluation is normalized to fit into the range [0, 1] (1 being best), and is called the *JARS evaluation*. We used all 118 Beans on JARS.COM which had been assigned a JARS evaluation in 2005 as an evaluation sample. Because Beans published on JARS.COM are used by a large number of people, it is reasonable to consider a high JARS evaluation as indication that the component has been reused a lot, and that the component quality, including its functional size, is good.

To determine a reference value for FOC, first the components were divided into the A group of 95 components with a JARS value of one, and the B group of 23 components with a JARS value of less than one. The average component FOC value was then computed for each group (shown in Table.1).

We adopted the A group average value shown in Table.1 as an appropriate upper-limit value for component functional size. Components whose measured FOC value is below this upper limit are considered to be of appropriate functional size.

Table.1: Average FOC values for both groups by category (N.b: number of beans, F.A/F.B: average FOC value of A/B group)

Category	N.b	F.A	F.B
Programming	113	204,717	2,771,720
Utilities	2	278,628	80
Game	2	-	200,950
Science	1	-	682,080
Average (total)	(118)	205,495	2,336,815

4.4 Visualization objects

In our system, the visual mapping displays elements for each of the visualization objects as follows.

- Components are visualized as boxes which reflect the FOC value of the component. The number of methods, events and properties are related to the width, depth and height of the box respectively, so that the volume of the box is a visual representation of the FOC, the component functional size. Then, the FOC value is evaluated against the FOC reference value to determine whether the component functional size is appropriate, and the object is displayed using a color that reflects this. If the FOC of a component is less than the FOC reference value, the box is displayed in blue, and if it exceeds the standard value the box is displayed in red.
- Java classes which are not components are displayed as light-green spheres.
- Dependency relationships are displayed as straight lines joining the visualization objects of the classes or components in the relationship.

As an example application of our system, we input some binary component software implemented using the "FukaBeans" JavaBeans binary component library [16]. Fig. 3 shows what this visualization looks like. As a result of analysis and visualization using our system, component and class characteristics as well as the how they are related can be visualized based on the dependency analysis and component functional size measurements. In Fig.3, the `FukaCalendar` component and the `FukaTextBean` component are differentiated from other classes, and displayed as blue boxes that reflect their functional sizes. In contrast, the `FukaCalendarExample` component is displayed as large red box in Fig.3. From this observation, it is found that our system help users to easily distinguish components with their functional sizes.

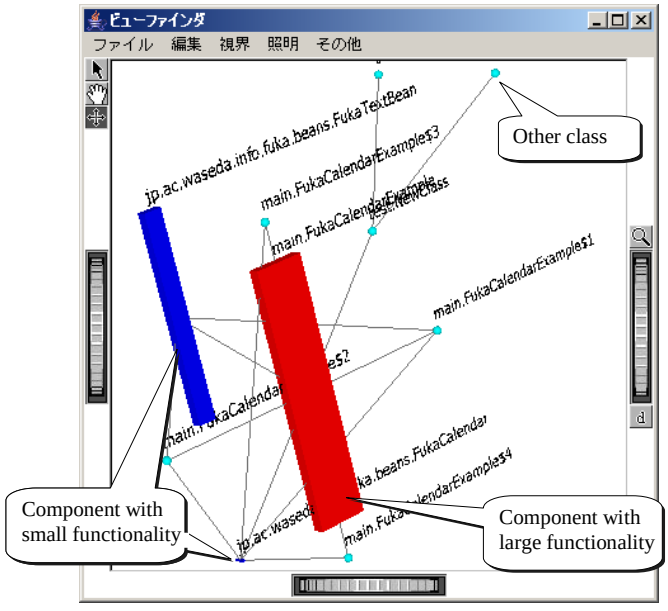

Fig.3: Example of visualization using our system

5 Comparative Evaluation

A comparison of the information that can be visualized by our system and two other systems, Seesoft and ObjectOrrery, (representing conventional static program visualization systems) is shown in Table.2. The table shows that the conventional visualization systems do not distinguish between classes and components that make up the program. They do not allow the user to visually differentiate them, and provide no support for an intuitive understanding of the internal structure. Further, analysis of the source code is a prerequisite for using these existing systems, so they cannot handle parts of binary component based program for which the source code is not available.

In contrast, our system differentiates between classes and components when displaying them, and displays the functional size of components visually. This supports the user in gaining an intuitive understanding of the internal structure of binary component based program.

Table.2: Comparison of visualization system features (Y: supported by visualization system, n: not supported)

Visualized data	Our system	Conventional system
Class	Y	Y
Component	Y	n
Functional size	Y	n
Inheritance relation	Y	Y
Reference relation	Y	Y
Instantiation relation	Y	Y

6 Conclusion

In this paper we have proposed a new system which analyzes and visually displays the structure of binary component-based program which includes JavaBeans components in binary form using reflection, introspection and byte-code analysis. The proposed system can facilitate an intuitive understanding of binary component-based program more effectively than existing visualization systems by clearly showing the functional size of components, by differentiating between classes and components, and by showing the dependency relationships between classes and components.

As a future work, we will extend our system to cover dynamic behavior of the target component-based program. Also, we plan to demonstrate clearly and conclusively the effectiveness of the system by using it in experiments to visualize and understand large-scale binary component-based program.

References:

[1] J.Q. Ning: A Component-Based Software Developement Model, 20th Annual International Computer Software and Applications Conference, pp.389-394, 1996.

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

[3] C. Atkinson: Component-based Product Line Engineering with UML, Addison-Wesley, 2001.

[4] H. Suzuki, et al.: Visualization of Program Behavior with Graphical Representation of Structure, 6th Interactive Systems and Software Workshop, pp. 99-104, 1998.

[5] S. Eick, J. Stefen, and E. Summer: Seesoft: A Tool for Visualizing Line Oriented Software Statistics, IEEE Transactions on Software Engineering, Vol.18, No.11, pp.957-968, 1992.

[6] N. Nishikawa: ObjectOrrery: 3d Visualization of Class Library Structure, 3rd Workshop on Software Visualization, 1999.

[7] G. Hamilton: JavaBeans 1.01 Specification, Sun Microsystems, 1997.

[8] J. Hopkins: Component Primer, Communications of the ACM, Vol.43, No.10, pp.27-30, 2000.

[9] H. Washizaki and Y. Fukazawa: A Technique for Automatic Component Extraction from Object-Oriented Programs by Refactoring, Science of Computer Programming, Vol.56, No.1-2, pp.99-116, 2005.

[10] H. Washizaki, H. Yamamoto and Y. Fukazawa: A Metrics Suite for Measuring Reusability of Software Components, 9th IEEE International Symposium on Software Metrics, pp.211-223, 2003.

[11] S. Chiba and M. Nishizawa: An Easy-to-Use Toolkit for Efficient Java Byte-code Translators, 2nd International Conference on Generative Programming and Component Engineering, pp.364-376, 2003.

[12] B.A. Myers: Visual Programming, Programming by Example and Program Visualization, ACM Conference on Human Factors in Computing System, pp.59-66, 1986.

[13] S.K. Card, et al.: Readings in Information Visualization, Morgan Kaufmann, 1999.

[14] Software Research Associates, Inc.: Jun for Java, `http://www.sra.co.jp/people/nisinaka/Jun4Java/`

[15] Jupitermedia Corporation: JARS.COM, `http://www.jars.com/`

[16] Waseda University, Fukazawa Laboratory: FukaBeans, http://www.fuka.info.waseda.ac.jp/Project/CBSE/