Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 56 (2005) 99–116

Science of Computer Programming

www.elsevier.com/locate/scico

A technique for automatic component extraction from object-oriented programs by refactoring

Hironori Washizaki[a,*], Yoshiaki Fukazawa[b]

[a] *Research Center for Testbeds and Prototyping, National Instituite of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan*

[b] *Department of Computer Science, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555, Japan*

Received 29 October 2003; received in revised form 22 September 2004; accepted 22 September 2004
Available online 21 December 2004

Abstract

Component-based software development (CBD) is based on building software systems from previously-existing software components. In CBD, reuse of common parts in component form can reduce the development cost of new systems, and reduce the maintenance cost associated with the support of these systems. However, existing programs have usually been built using another paradigm, such as the object-oriented (OO) paradigm. OO programs cannot be reused rapidly or effectively in the CBD paradigm even if they contain reusable functions. In this paper, we propose a technique for extracting components from existing OO programs by our new refactoring “Extract Component”. Our technique of refactoring can identify and extract reusable components composed of classes from OO programs, and modify the surrounding parts of extracted components in original programs. We have developed a system that performs our refactoring automatically and extracts JavaBeans components from Java programs. As a result of evaluation experiments, it is found that our system is useful for extracting reusable components along with usage examples from Java programs.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Component-based development (CBD); Refactoring; Object-oriented programming; Software reuse; Software component; JavaBeans

* Corresponding author.
E-mail addresses: washizaki@acm.org (H. Washizaki), fukazawa@waseda.jp (Y. Fukazawa).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.11.007

1. Introduction

Component-based software development (CBD) has become widely accepted as a cost-effective approach to software development [35]. In CBD, software development is considered to involve the composition of various software components. CBD is capable of reducing developmental costs and improving the reliability of an entire system.

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 [13]. In fact, some of the practical component architectures, such as JavaBeans [12] and Enterprise JavaBeans (EJB) [5], are based on OO technologies.

In CBD, the reuse of common parts in component form can reduce the development cost of new systems, and reduce the maintenance cost associated with the support of these systems. However, not all components corresponding to functional requirements are already available in all possible contexts. In contrast, there are many program repositories available on the Internet such as SourceForge.net [34]. At these on-line repositories, programmers can obtain a large amount of OO program source codes and binary codes. There is a possibility that programs, which partially fulfill the required functionalities, exist among these available OO program source codes. If such parts of existing OO programs could be easily reused as components, programmers could develop software by means of CBD by utilizing these programs.

However, since OO classes usually have complex mutual dependencies, it is difficult to reuse parts of existing OO programs composed of classes rapidly and effectively. If a significant function is realized by a set of classes, programmers who want to reuse the function must examine the dependencies among related classes and acquire all depending classes. Since such manual examination activities entail a high cost for programmers, the merits of reuse might be reduced or lost. Therefore, it is necessary to transform a part of an existing OO program into a component that has no dependence on elements outside itself. However, current CBD methodologies mostly lack a systematic decomposition algorithm [33].

Moreover, even if the components can be extracted from existing programs, it is difficult to identify the appropriate use of the extracted components only by referring to the source codes or public interfaces of the components. Therefore, it is preferable to acquire a usage example along with the extracted components.

In this paper, we propose a technique for identifying structurally reusable candidate parts of OO programs according to our definition of the reusable component based on JavaBeans [12], and transforming these parts into reusable components automatically by our new refactoring, “Extract Component”. Our technique targets Java language as the OO programming language and JavaBeans as the fundamental component architecture. Our technique accepts any kind of Java programs as the extraction target whether these programs use specific coding standards or structural templates. Moreover, we show that our extraction technique is useful for acquiring usage examples for the extracted components.

In the following, we first define a class relation graph (CRG) that represents the relations among classes/interfaces in the target Java program. Next, using a CRG, we propose a technique for extracting components from OO programs, and changing the parts surrounding the extracted components to allow these surrounding parts to use the newly extracted components. These surrounding parts become the usage examples of the extracted components.

2. Class relation graph

To extract reusable components from Java programs, we first define a CRG. The CRG is obtained by a static analysis of the dependencies among Java classes/interfaces. Next, we provide a clustering algorithm to detect all possible clusters by determining the reachability on the given CRG. A cluster is a candidate component, and has no dependence on elements outside the cluster.

In the following, classes in Java core packages (e.g., java.*, javax.*) do not become nodes in the CRG, because the core packages are distributed by standard Java runtime systems. We consider only programmer-made classes/interfaces.

Definition 1 (*Class Relation Graph*). The multigraph which satisfies all the requirements, 1–7, is a CRG, denoted as $\Gamma = (V, \Lambda, E)$, where V is a set of Java class/interface nodes, Λ is a set of sequential numbers that are used for the identification of edges, and E is a set of directed edges that are ordered trios of a source node, a destination node, and a label name.

1. $V = VC \cup VI$

VC is the set of class nodes corresponding to classes. *VI* is the set of interface nodes corresponding to interfaces. All nodes are represented as rectangles that have class/interface names inside when the CRG is illustrated in the form of a figure. In the following, a node (on the CRG) that corresponds to class or interface c is described as $Node(c)$, while a class or interface that corresponds to node v on the CRG is described as $Node^{-1}(v)$.

2. $E = EE \cup EI \cup ER$

E consists of a set of inheritance edges (*EE*), a set of instantiation edges (*EI*), and a set of reference edges (*ER*).

3. $EE \subseteq VC \times V \cup VI \times VI$

EE is a set of inheritance edges, which indicate that a class inherits another class/interface, or an interface inherits another interface, and is denoted as $-\triangleright$.

4. $EI = EID \cup EIA$

EI consists of a set of default instantiation edges (*EID*) and a set of argument instantiation edges (*EIA*). The instantiation edge is drawn from a class instantiating another class to the target class for instantiation. Each instantiation edge contains an unique label for identification.

5. $EID \subseteq V \times VC \times \Lambda$

EID is a set of default instantiation edges, which indicate that a class/interface instantiates an object of another class by using a default constructor (a constructor without any

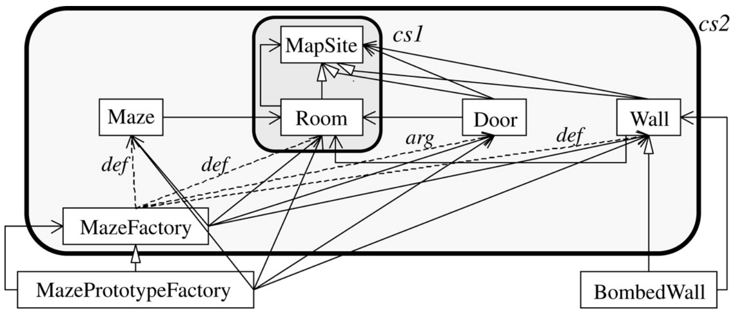


Fig. 1. CRG of prototype pattern's code.

arguments) of the target class for instantiation, and is denoted as $\overset{def}{\cdots>}$.

6. $EIA \subseteq V \times VC \times \Lambda$

EIA is a set of argument instantiation edges, which indicate that a class/interface instantiates an object of another class by using a constructor with one or more arguments, and is denoted as $\overset{arg}{\cdots>}$. At this time, the primitive types (e.g., int) and classes in the Java core packages do not become the target of the above-mentioned "arguments". For example, if the class c instantiates an object of another class c' by using the c''s constructor whose type of argument is int, we denote this as $Node(c) \overset{def}{\cdots>} Node(c')$.

7. $ER \subseteq V \times V \times \Lambda$

ER is a set of reference edges, which indicate that a class/interface refers to another class/interface with a unique label for identification, and is denoted as $\rightarrow$. In this paper, we (and our extraction system described in Section 5.1) recognize that a class/interface c_a refers to another class/interface c_b, when there is a type specification of c_b for the variable/field/method declarations in c_a, or there is a type specification of c_b for the type casts in c_a, or c_a accesses a method/field of c_b. In addition, we also recognize that an outer class/interface refers to its inner class/interface in the same source code.

As an example of analyzing a small program to which the typical OO design has been applied, Fig. 1 shows the CRG of the Prototype design pattern [9] sample code written in Java language [10] with some code modifications. Fig. 2 shows all modified parts of the original code. This sample consists of seven classes and one interface. In this case, the CRG is composed of the following elements:

$$\begin{aligned}
VC &= \{\, \texttt{BombedWall}, \texttt{Door}, \texttt{Maze}, \texttt{MazeFactory}, \texttt{MazePrototypeFactory},\\
&\quad \texttt{Room}, \texttt{Wall} \,\}\\
VI &= \{\, \texttt{MapSite} \,\}\\
EE &= \{\, (\texttt{BombedWall},\texttt{Wall}), (\texttt{Door},\texttt{MapSite}), (\texttt{Room},\texttt{MapSite}),\\
&\quad (\texttt{Wall},\texttt{MapSite}), (\texttt{MazePrototypeFactory},\texttt{MazeFactory}) \,\}\\
EID &= \{\, (\texttt{MazeFactory},\texttt{Maze},9), (\texttt{MazeFactory},\texttt{Wall},10),\\
&\quad (\texttt{MazeFactory},\texttt{Room},11) \,\}
\end{aligned}$$

```
public class MazeFactory {
 public Maze makeMaze(){ return new Maze();}
 public Wall makeWall(){ return new Wall();}
 public Door makeDoor(Room r1, Room r2) { return new Door(r1, r2); }
 public Room makeRoom(int n) {
  Room room = new Room(); room.roomNumber = n; return room; }
}

public class Room implements MapSite, Cloneable {
 public int roomNumber;
 private MapSite sides[] = new MapSite[4];
 public Room(){ }
 public Room(int r) { roomNumber = r; }
 public MapSite getSide(int aDirection) { return sides[aDirection]; }
 public void enter() { ... }
 public void initialize(int n) { ... }
 public Object clone(){ try{ return super.clone(); }
  catch(Exception e){...} }
}
```

Fig. 2. Modified parts of original code.

$$EIA = \{\, (\texttt{MazeFactory},\texttt{Door},12)\,\}$$
$$ER = \{\, (\texttt{BombedWall},\texttt{Wall},0), (\texttt{BombedWall},\texttt{Room},1), (\texttt{Door},\texttt{MapSite},2), \ldots \}.$$

The main functionality of this sample is making a maze. Moreover, this sample provides common constructs of the maze, such as Room and Door. If programmers can reuse the functionality or constructs in component form, a new program that makes another but similar maze can be easily developed.

Next, we define several reachabilities on the CRG in order to provide a clustering algorithm for the identification of structurally reusable candidate parts from among a collection of classes/interfaces.

Definition 2 (*Reachability*). In an arbitrary CRG, node v is inheritance-reachable (instantiation-reachable) from node u when there is a directed path from u to v and all edges on the path are inheritance edges (instantiation edges), or $u = v$, denoted as $u \overset{*}{-\!\!\triangleright} v$ ($u \overset{*}{\cdots>} v$). Similarly, node v is dependence-reachable from node u when there is a directed path from u to v and all edges on the path are inheritance/instantiation/reference edges, or $u = v$, denoted as $u \overset{*}{\Rightarrow} v$. In addition, we describe that $Node^{-1}(v)$ is reachable from $Node^{-1}(u)$ when v is reachable from u in a corresponding CRG.

For example, in Fig. 1, there are two elements (MapSite and Room) that are dependence-reachable from Room.

3. Software components and their usage examples

In the case of using OO programming languages for the implementation of components, a component is structurally defined as a reusable/substitutable set of OO classes. However, this definition does not clarify the requirement for the component's substitutability. The structural units for programmers are a class and a set of classes. Among various sets of classes, there is a set that has no hotspots (necessary for OO frameworks) and that can be manipulated in a uniform way. Such a set of classes can be recognized as a component on the basis of a particular component architecture that provides a uniform means of using the set of classes.

In this paper, we target JavaBeans as a component architecture. In the following, we first refer to the default definition of JavaBeans components. Then, we provide a strict definition of the reusable component based on JavaBeans. The reusable component, which satisfies the definition given below, has no dependence on elements outside of itself, and can be instantiated and used alone.

3.1. Definition of reusable component

Definition 3 (*Reusable Component*). A reusable component is a set of Java classes/interfaces that satisfies all the following requirements, (1)–(4), and is packaged into one JAR file.

(1) The classes can be treated as a JavaBeans component.

To be treated as a JavaBeans component, there are four requirements for the target set of classes. First, the set must include one Facade class, which plays the role of the facade for the outside of the classes based on the Facade design pattern [9]. Second, the Facade class must have one public default constructor (a public constructor without any arguments). Third, the Facade class must implement the `java.io.Serializable` interface because the Facade class must be serializable to enable saving and restoring of the state of the component using the Java Serialization APIs. Fourth, the set preferably should be packaged as one JAR file including a manifest file that explicitly specifies the Facade class.

However, requirement (1) allows components to depend on classes that are not included in a component's JAR file. In other words, JavaBeans components can have external dependencies. Moreover, this requirement does not provide any means of using the components' functions without an understanding of the content of the components. This leads to the problem that a JavaBeans component without any additional restrictions is not reusable in all possible contexts. Therefore, we add the following requirements, (2)–(4), to ensure that the target component is indeed reusable.

(2) An interface (Facade interface) declaration of the Facade class is separated from its implementation.

All method invocations from outside a component must be realized via the Facade interface. This separation ensures that any implementation change for the same interface does not influence the client sides of the component. Moreover, since the internal structure is hidden from outside the component, a user need only understand the Facade interface.

(3) All classes/interfaces necessary for instantiating an object of the Facade class must be packaged into a component's JAR file.

The entire component must be instantiated by invoking a default constructor of the Facade class. To enable this, all classes/interfaces necessary for instantiating an object of the Facade class must be distributed to the user. This requirement excludes the possibility that the component's participant classes access any class outside the component.
(4) All classes in the set, except static classes, must be instantiation-reachable from the Facade class.

Definition 4 (*Static Class*). A static class is a class of which all fields and methods are declared with the modifier "`static`". In the following, the expression *Static*(c) is true if class c is a static class.

Objects of all the component's participant classes, except static classes, must be directly or indirectly instantiated by the Facade class. This requirement reduces the necessity of instantiating and passing objects of the component's participant classes from the outside of the component to the component. Therefore, for users who want to reuse components, the necessity of understanding the content of components reduces.

For simplicity of description, "component" means the reusable component in the following. The component based on the above-mentioned definition does not always represent a semantically reusable component in possible contexts. However, the component is structurally reusable, because the internal structure is hidden from outside, and the component can be instantiated/executed independently in other contexts.

3.2. Restriction on target program

As mentioned earlier, it is preferable that the usage example of a component can be obtained when extracting the component from a program. In this paper, we treat the program, which instantiates an object of the component's Facade class and uses the component via the Facade interface without instantiating any other objects of the component's participant classes, as the usage example. The usage example of the extracted component helps programmers reuse the component in an appropriate way.

If the target program for extraction satisfies the following two constraints, programmers can acquire a usage example that appropriately uses the set of Java classes/interfaces (which satisfies all the above-mentioned requirements) as a component when extracting the set.

- One or more classes outside of the set have instantiated an object of the Facade class of the set by using a default constructor of the Facade class. This means that the object of the Facade class must be instantiated at least once from outside the component.
- Objects of classes in the set, except the Facade class, have not been instantiated by classes outside of the set. This means that no classes that compose the extracted component should be treated in any way other than as elements of the component.

4. Component extraction

Using the CRG, the necessary procedures for extracting components and acquiring usage examples are shown below. The component based on our definition of the reusable component cannot be extracted by simply grouping dependent OO classes/interfaces.

The extraction procedures require more complex operations, such as detecting candidates of Facade classes and creating Facade interfaces.

4.1. Extraction algorithm

Our technique detects all clusters that are candidates of components in the CRG, using the following clustering algorithm.

Definition 5 (*Component Cluster*). The pair of the Facade node $Node(c)$ and the set of nodes V_c is a cluster, denoted as $cs = (Node(c), V_c)$, in which all of the following requirements are satisfied. The cluster in the CRG is a candidate for a component.

- Class c is a concrete class that has a default constructor or a constructor with only primitive types or classes in Java core packages as its arguments (called a "basic constructor"). This requirement is expressed as $Default(c)$.
- V_c is a set of all nodes that are dependence-reachable from $Node(c)$.
- All nodes in V_c, except the nodes corresponding to static classes, are instantiation-reachable from $Node(c)$.

Definition 6 (*Clustering Algorithm*). The following clustering algorithm $\mathrm{GC}(u, \Gamma)$ specifies the set of all possible clusters ($S = \{cs_1, \ldots, cs_m\}$) using a given node u as the starting node for exploration. The starting node is an extraction criterion.

```
GC($u, \Gamma$)
    $S \leftarrow \Phi$;
    $V_c \leftarrow \{v \mid (u \overset{*}{\Rightarrow} v)\}$;
    for_each $v_f \in V_c$ do
        if $Default(Node^{-1}(v_f))$
            $\wedge\, v_f \overset{*}{\Rightarrow} u \wedge (\forall v' \mid v' \in V_c \Rightarrow (v_f \overset{*}{\cdots>} v' \vee Static(Node^{-1}(v'))))$
        then
            $S \leftarrow S \cup \{(v_f, V_c)\}$;
        end_if
    end_for
    return $S$;
End_of_GC
```

Definition 7 (*Determination of Usage Example Acquisition*). The following boolean expression $\mathrm{VC}(cs, \Gamma)$ for obtained cluster $cs = (v_f, V_c)$ and the CRG Γ determines whether the parts surrounding a component corresponding to cs become the usage example of the component in the target program corresponding to Γ. If VC is true, programmers can use the target program for extraction as the usage example of the newly extracted component.

$$\mathrm{VC}(cs, \Gamma) \equiv (\exists v \mid v \in V - V_c \Rightarrow v \overset{def}{\cdots>} v_f) \wedge \neg(\exists u \mid u \in V - V_c \Rightarrow (\exists w \mid w \in V_c - \{v_f\} \Rightarrow u \overset{def}{\cdots>} w \vee u \overset{arg}{\cdots>} w)).$$

4.2. Extract component refactoring

When extracting components corresponding to detected clusters after using the clustering algorithm, the parts surrounding the clusters should be modified to use newly extracted components in order to avoid the situation where two sets of classes that provide the same function exist in the same organization/library. At this time, the surrounding parts become the usage example of the extracted components.

This modification is a kind of refactoring because this modification does not change the observable behavior of the original target program for extraction. Refactoring is a technique for changing the internal structure of existing programs to make them easier to understand and cheaper to modify, without changing their observable behavior [29,7]. We call this refactoring "Extract Component", and all necessary steps for this refactoring are shown below.

(1) Make a CRG, $\Gamma = (V, \Lambda, E)$, of the target Java program.

(2) As the extraction criterion, select a starting node $u \in V$ corresponding to a class which provides the functions that you want to reuse.

(3) Obtain all clusters that contain u by applying $\mathrm{GC}(u, \Gamma)$ to Γ.

(4) Select one cluster $cs = (v_f, V_f)$ from the obtained clusters $S = \mathrm{GC}(u, \Gamma)$. In the following, class $Node^{-1}(v_f)$ is described as c_f.

(5) Create a new Facade interface i_f whose name is "`I`" $+\langle c_f$'s name$\rangle$. Implement i_f to c_f.

(6) Add the declarations of all public methods implemented within the classes (C_E), which are inheritance-reachable from c_f ($C_E = \{Node^{-1}(v) \mid (v_f \overset{*}{\rightarrow\!\!\triangleright} v) \text{ in } \Gamma\}$), into i_f.

(7) Add the declarations of the setter methods and getter methods corresponding to all public fields of C_E into i_f. However, only the getter methods must be added if the corresponding fields are declared with the accessible modifier "`final`". The setter methods must be named "`set`" $+\langle$field's name$\rangle$. The getter methods must be named "`get`" $+\langle$field's name$\rangle$.

(8) Add the implementations of the setter methods and getter methods, which are newly declared within i_f at Step 7, into c_f. This step is a kind of Encapsulate Field refactoring [7]. The setter methods must be simply implemented to change the value of the corresponding field using an input value. The getter methods must be simply implemented to return the value of the corresponding field.

(9) If c_f has been used for the types of the method arguments, method return values, or throwable exceptions, change these types from c_f to i_f.

(10) Implement the `java.io.Serializable` interface to c_f.

(11) Set the protection modifier of c_f to "`public`".

(12) If c_f has a default constructor, set the protection modifier of the default constructor to "`public`". If c_f has only a basic constructor, add a new public default constructor, which invokes the basic constructor using the initial values of argument types for its arguments, into c_f. The initial value of each type is uniquely fixed in the following way.

- If the type is a primitive type, the value when only a variable of this type is declared becomes the initial value. For example, the initial values of `int` and `boolean` are zero and `false`, respectively.
- If the type is a class in Java core packages, the value when only a public default constructor of this type is invoked for instantiation of its object becomes the initial value. If the public default constructor is not available, attempt to obtain the initial value by an invocation of a constructor, whose number of arguments is the least among all constructors, using initial values corresponding to types of constructor arguments recursively.
- Otherwise, `null` is used as the initial value.

(13) By using a Java compiler, compile the source codes of modified c_f and newly created i_f.

(14) Create a manifest file, which specifies c_f. Package all class files of classes/interfaces in V_f and i_f into one JAR file. The name of c_f becomes the component's name.

Next, if the expression $\mathrm{VC}(cs, \Gamma)$ is true, execute the following steps, 15–17.

(15) Change the program codes, which assign new values to fields of V_E via the reference type for c_f (or refer the values of fields of V_E) in $V - V_f$, to the program codes that invoke the setter methods (or getter methods) of c_f.

(16) If the implicit widening reference conversion from c_f to another class/interface c_e in V_E exists in $V - V_f$, insert the explicit reference conversion program code, which converts the reference types from c_f to c_e, into all parts where the implicit conversion exists.

(17) Change the reference types, which refer to c_f in $V - V_f$, to those that refer to i_f.

By means of the above-described procedure, programmers can extract components that can be instantiated independently of other classes/interfaces, and that can be reused via the Facade interface in other contexts. Moreover, if the determination algorithm determines that the usage example can be acquired, programmers can use the entire modified program for extraction as a usage example that appropriately uses the newly extracted component.

For example, we obtained two clusters by applying $\mathrm{GC}(u, \Gamma)$ to the CRG Γ shown in Fig. 1 using all nodes as starting nodes ($u \in V$):

$$cs1 = (\mathtt{Room}, \{\mathtt{Room}, \mathtt{MapSite}\}),$$

$$cs2 = (\mathtt{MazeFactory}, \{\mathtt{MazeFactory}, \mathtt{Room}, \mathtt{MapSite}, \mathtt{Door}, \mathtt{Maze}, \mathtt{Wall}\}).$$

By creating Facade interfaces, the component corresponding to $cs1$ will encapsulate a common construct of the maze, and the component corresponding to $cs2$ will encapsulate the functionality of making a maze. It is thought that these components can be reused in the context of developing another maze program.

Among these clusters, $\mathrm{VC}(cs1, \Gamma)$ becomes true, and $\mathrm{VC}(cs2, \Gamma)$ does not. Therefore, a usage example of a component can be acquired when extracting the component corresponding to $cs1$. As a result of the Extract Component refactoring, Fig. 3 shows a UML class diagram of a newly extracted component (named "Room") that corresponds to $cs1$.

Fig. 3 shows a newly created Facade interface "`IRoom`", which is implemented by the Facade class "`Room`". Moreover, all class files of classes/interfaces (`Room`, `IRoom`,

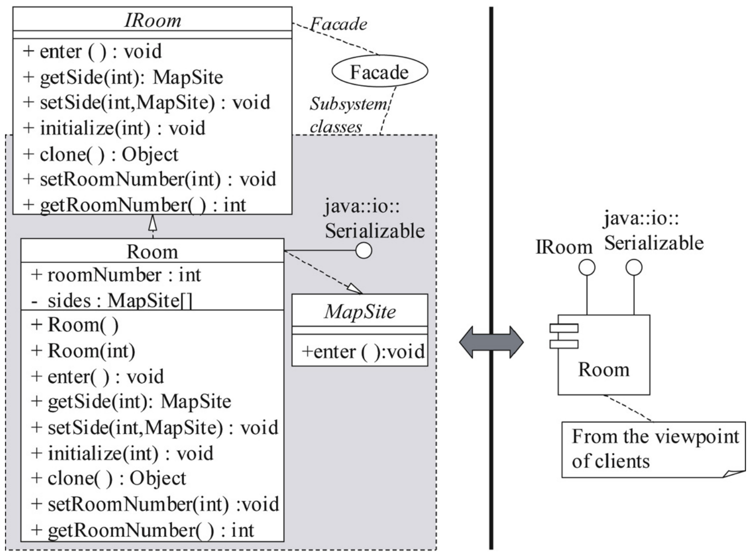


Fig. 3. UML diagram of extracted component.

and `MapSite`) necessary for instantiating an object of the Facade class have been packaged into one JAR file as a component. `IRoom` has designated the getter method and the setter method corresponding to `Room`'s public field "`roomNumber`", and implementations of these getter/setter methods have been added to `Room`. Since the extracted component is independent of all other classes/interfaces in the original program, programmers can reuse this component via the Facade interface in other contexts.

Fig. 4 shows a part of the changed sample code after the component corresponding to *cs*1 has been extracted from the sample code shown in Fig. 2. In `MazeFactory` and `MazePrototypeFactory`, the reference types that referred to `Room` have been changed to the reference types that refer to `IRoom`. Moreover, in `MazeFactory`, the program code accessing the public fields of `Room` has been changed to a program code that uses the setter method, which is newly declared in `IRoom`. With these modifications, the extracted component is now used via `IRoom` by the component's clients. At this time, `MazeFactory` has become a usage example that instantiates an object of the Facade class of the component and uses the component via the Facade interface.

5. Automation system and experimental evaluation

5.1. Automatic extraction system

Manually transforming existing programs into components incurs considerable costs. Some of existing refactoring tools, such as RefactorIT [1] and JRefactory [19], can support a part of the necessary steps of our refactoring (e.g. (5)–(8)); however, the overall steps of the refactoring should be supported by just one tool in order to perform the refactoring more smoothly and correctly.

```
public class Room implements IRoom, MapSite,Cloneable,  // Changed
                             java.io.Serializable     { // Changed
 ...
 public void setRoomNumber(int x) { roomNumber = x; }   // Added
 public int getRoomNumber(){ return roomNumber; }       // Added
}

public class MazeFactory { ...
 public IRoom makeRoom(int n) {                         // Changed
  IRoom room = new Room();                              // Changed
  room.setRoomNumber(n);                                // Changed
  return room; }
}

public class MazePrototypeFactory extends MazeFactory { ...
 private IRoom prototypeRoom;                           // Changed
 public IRoom makeRoom(int n) {                         // Changed
  IRoom room = (IRoom) prototypeRoom.clone();           // Changed
  room.initialize(n);
  return room; }
}
```

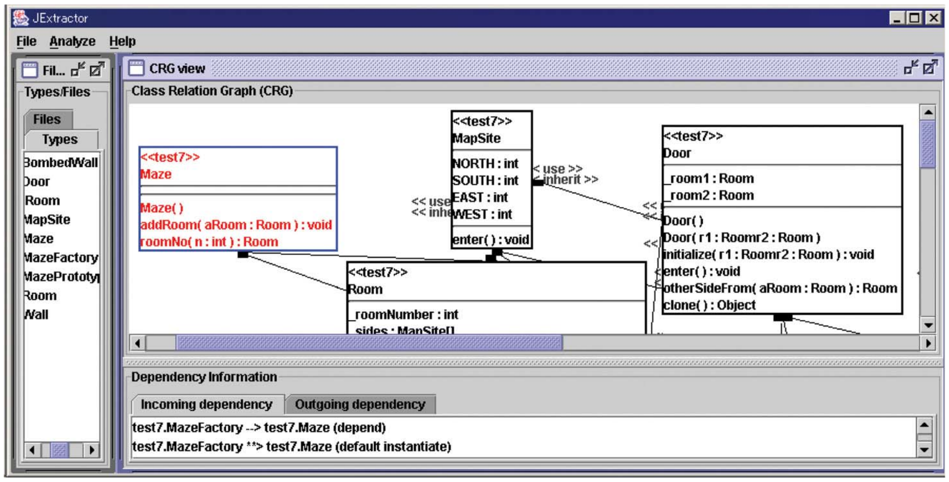
Fig. 4. Part of program code after component extraction.

Therefore, we have developed an automatic component-extraction system in Java language, using JavaCC [17] as a parser generator. Our system executes automatically all the necessary steps of the Extract Component refactoring, except the selection steps of the starting node and obtained cluster. In Fig. 5, our system has analyzed the Prototype pattern sample code and displayed a CRG. Using the graphical user interface of our system, programmers can visually confirm the CRG, select a class corresponding to the starting node, initiate execution of the Extract Component refactoring, and obtain changed program codes and the extracted JavaBeans component.

5.2. *Experimental evaluation*

We have applied our system to nine Java programs actually used. The evaluated programs are JUnit [20] (total number of classes/interfaces including inner classes: 181), JBeans [18] (241), Regexp [30] (16), JXPath [21] (121), TableExample [16] (18), Metalworks [16] (32), SampleTree [16] (18), Font2DTest [16] (20), and FileChooser [16] (8). No classes in these programs have been packaged as JavaBeans components. We have attempted to extract all possible components from these four programs, using all classes in the programs as classes corresponding to starting nodes.

Table 1 shows the number of extracted components (denoted as N.All), and the average value of the number of participant classes/interfaces except the newly created Facade



```
test7.MazeFactory --> test7.Maze (depend)
test7.MazeFactory **> test7.Maze (default instantiate)
```

Fig. 5. Screen shot of extraction system.

interface in each component (#V_c) for each program. Table 1 also shows the number of extracted components that have usage examples, as determined by VC (N.Ex).

Moreover, Table 1 shows the number of components for which the measurement value of the reusability metric DIT [3] or SCCr [37] lies within its confidence interval (N.DIT, N.SCCr), and the number of components for which at least one measurement value of these reusability metrics lies within the corresponding confidence interval (N.D∪S).

Definitions of these reusability metrics are shown below.

- Depth of Inheritance Tree (DIT [3]) for the Facade class: the confidence interval is [lower confidence limit: 2, upper confidence limit: 4]. DIT for the Facade class is the maximum number of steps from the Facade class to the root of the inheritance tree.
- Self-Completeness of Component's Return Value (SCCr [37]): the confidence interval is [0.61, 1.0]. SCCr is the percentage of methods without any return value in all methods, except the getter and setter methods, implemented within the component.

We previously confirmed that a component's reusability is high if at least one of the measurement values of these two metrics lies within the corresponding confidence interval [37]. By an interval estimation based on the nonparametric test statistic [25], we calculated the confidence intervals with a confidence coefficient of 95% for these reusability metrics using all JavaBeans components that were judged to be highly reusable by JARS.COM [15].

As shown in Table 1, our system has extracted many components, which consist of one or more classes/interfaces, from four programs regardless of the kind of program.

Fig. 6 shows the histograms of the number of participant classes/interfaces, except the Facade interface, in each component for all programs. We found that 52.9% of all extracted components consist of the Facade class and the Facade interface (#$V_c = 1$), and 47.1% consist of three or more classes including the Facade interface (#$V_c \geq 2$). This result means that half of all extracted components have been acquired by gathering more than one

Table 1
Number of extracted components for each program

Program	N.All	N.Exa	N.DIT	N.SCCr	N.D∪S	$\#V_c$
JUnit	11	6	3	7	8	1.5
JBeans	16	9	5	10	12	1.4
Regexp	9	2	1	2	2	4.8
JXPath	22	13	5	7	9	2.6
Metalworks	16	14	6	14	14	5.3
TableExample	5	1	2	2	4	2.0
SampleTree	3	1	1	2	2	7.0
Font2DTest	2	1	0	2	2	19.0
FileChooser	3	2	2	2	3	3.3
Total (Average)	87	49	25	48	56	(3.47)

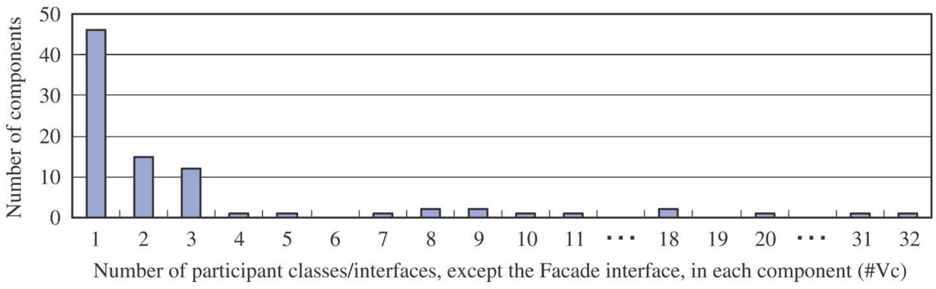


Fig. 6. Histograms of the number of participant classes/interfaces in each component.

depending classes/interfaces in the original programs. Moreover, even if each of all classes in the original program can be reused independently, our system is still useful because our system automatically detects such independence in the classes and correctly extracts a minimum size component.

Therefore, we note that our approach of detecting and packaging all depending classes/interfaces into one component automatically has been effective. According to the definition of the reusable component, it is guaranteed that each extracted component has no dependence on elements outside of itself, and can be instantiated and used alone.

From the measurement values of reusability metrics, it is found that almost 2/3 ($100 \times$ N.D∪S / N.All $= 64.4\%$) of all extracted components are highly reusable. This result suggests that our system is useful for extracting reusable components from existing Java programs. For example, from JUnit, we extracted eight components for which at least one measurement value of the reusability metrics lies within the confidence interval. The Facade classes of these components are `StatusLine`, `awtui.ProgressBar`, `AssertionFailedError`, `Assert`, `Sorter`, `NoTestCaseClass`, `swingui.ProgressBar`, and `TestCaseClassLoader`. Many of these components provide the reusable functions, such as the sorting values, displaying progress bar, and enabling assertions. This result originates in the fact that the content of JUnit has been carefully designed to avoid high coupling between classes that do not relate.

In Table 1, more than half ($100 \times$ N.Exa / N.All $= 56.3\%$) of all extracted components have usage examples. In other words, the target programs for extraction in these experiments can become the usage examples for the majority of extracted components. This result suggests that our system is useful for acquiring the usage example along with components when extracting the components.

6. Related work

Our refactoring technique is the first one to reuse existing OO programs by automatically extracting reusable components along with usage examples. Nonetheless, our approach bears resemblance to several existing techniques: component extraction, software clustering, and program/interface slicing.

6.1. Component extraction

There are several techniques for extracting components from object-oriented program codes or models.

Song has proposed an analysis technique for extracting EJB components from Java servlet programs [33]. However, the target of this technique has been limited to the Java servlet. In contrast, the target of our technique is any kind of Java program.

Lorenz has proposed an architectural technique for transforming Java classes into JavaBeans components [27]. This technique aims to map one class to one component and map one method to multiple of events. This results in an excessive number of components for existing classes. In contrast, our technique aims to map multiple classes to one component.

Lee proposes a component identification technique that can be applied to UML descriptions based on the class relation graph [24]. However, the necessary steps for identification are not automated.

6.2. Software clustering

Software clustering attempts to decompose software systems into meaningful subsystems to facilitate understanding of those systems or to reuse subsystems. Conventional clustering techniques can be classified as follows according to the kind of information they use [22]: domain-model-, dataflow-, connection-, metric-, and concept-based approaches. However, none of these conventional techniques guarantees that result set of subsystems have no dependence on elements outside of each subsystem and can be instantiated/executed as standalone components.

- Domain-model-based approaches use an object model that describes the object candidates and their relationships [8]. The model is used to recover an architectural view of the target procedural codes.
- Dataflow-based approaches use dataflow information to identify object candidates and transform procedural programs into object-oriented programs [36].

- Connection-based approaches cluster entities (variables, types, objects, classes, or subprograms) based on a specific set of directed relationships between entities to be grouped [4,11]. For example, Girard has performed the dominance analysis of the relation on the call graph to group functions/variables into modules and subsystems as component candidates [11].
- Metric-based approaches cluster entities based on a specific metric. Hutchens's approach groups related subprograms using a similarity metric based on data bindings, which correspond potential data exchange among global variables in programs [14]. Doval's approach uses a genetic algorithm to determine "good" partitions of the target system [6]. The algorithm partitions the system into highly cohesive subsystems such that these subsystems are loosely coupled together.
- Concept-based approaches use the concept analysis to obtain a lattice of concepts, which are maximal sets of objects sharing common attributes [26,32]. The obtained lattice can be composed of separate sublattices as component candidates.

Visualization tools for interactive reverse engineering, such as Rigi [28] and Understand for Java [31], can be used to present an architecture of the target program and to help the acquisition of dependent classes as subsystems. However, users of these tools must still perform additional tasks to package dependent classes into one executable component. Moreover, when two or more related classes are obtained, users must judge which class plays the role of controlling other obtained classes.

6.3. Slicing

A technique for program slicing can extract executable slices from existing OO programs [23]. Using this technique, programmers can obtain a fine-grained slice composed of related program statements. However, since it is necessary to specify a certain variable in the program for slicing, this technique assumes that the user knows the detail of the target program. Therefore, this technique cannot be appropriately applied to OO programs provided by third parties. In contrast, our technique does not require users to know the detail (e.g. role/intention of each variable) of the third-party programs.

Beck has proposed a module-level slicing technique, called interface slicing, to slice a subset of the behavior of a single module [2]. However, since this technique does not concern the relation among modules, slices cannot be composed of operations that belong to different modules.

7. Conclusion and future work

We have proposed a component extraction technique described as Extract Component refactoring. Our refactoring identifies the structurally reusable candidate parts of Java programs, transforms these parts into reusable JavaBeans components automatically, and modifies the parts surrounding the obtained components in the original programs in order to acquire the usage example. Moreover, we have developed a system that statically analyzes Java program source codes and executes all steps necessary for our refactoring automatically.

As a result of experimental evaluations, we have confirmed that our system can extract many highly reusable components from Java programs actually used. We have also confirmed that the target programs for extraction can become the usage examples for the majority of extracted components. This result suggests that our system is useful for extracting reusable components along with usage examples from existing Java programs.

Our system is currently developed to accept Java program source codes and extract JavaBeans components only. However, the clustering method of our technique can be similarly applied to other component architectures that use statically typed OO languages for components' implementation. When applying our technique to other architectures, it is necessary to change a part of the procedure of our extraction refactoring (especially (5)–(14)) according to the component's definition of the selected architectures. We will extend our system to accept source codes in other programming languages, and extract components based on other component architectures, such as C++ and ActiveX.

References

[1] Agris Software AS, RefactorIT: Java Refactoring Tool, http://www.refactorit.com/.
[2] J. Beck, D. Eichmann, Program and interface slicing for reverse engineering, in: Proc. 15th International Conference on Software Engineering, 1993, pp. 509–518.
[3] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[4] A. Cimitile, G. Visaggio, Software salvaging and the call dominance tree, Journal of Systems Software 28 (1995) 117–127.
[5] L.G. DeMichiel, Enterprise JavaBeans 2.1 Specification, Sun Microsystems, 2003.
[6] D. Doval, S. Mancoridis, B.S. Mitchell, Automatic clustering of software systems using a genetic algorithm, in: Proc. International Conference on Software Tools and Engineering Practice, 1999, pp. 73–81.
[7] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[8] H. Gall, R. Kosch, J. Weidl, Resolving uncertainties in object-oriented re-architecting of procedural code, in: Proc. 7th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, 1998, pp. 726–732.
[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (S. Honiden, K. Yoshida Trans.), Softbank Publishing, 1999 (in Japanese).
[11] J.F. Girard, R. Koschke, Finding components in a hierarchy of modules: a step towards architectural understanding, in: Proc. 13th International Conference on Software Maintenance, 1997, pp. 66–75.
[12] G. Hamilton, JavaBeans 1.01 Specification, Sun Microsystems, 1997.
[13] J. Hopkins, Component primer, Communications of the ACM 43 (10) (2000) 27–30.
[14] D.H. Hutchens, V.R. Basili, System structure analysis: clustering with data bindings, IEEE Transactions on Software Engineering 11 (8) (1985) 749–757.
[15] JARS.COM, http://www.jars.com/.
[16] Java2SDK1.4.1 demo, http://java.sun.com/.
[17] Java Compiler Complier, http://javacc.dev.java.net/.
[18] JBeans, http://jbeans.sourceforge.net/.
[19] JRefactory, http://jrefactory.sourceforge.net/.
[20] JUnit, http://junit.sourceforge.net/.
[21] JXPath, http://jakarta.apache.org/commons/jxpath/.
[22] R. Koschke, An incremental semi-automatic method for component recovery, in: Proc. 6th Working Conference on Reverse Engineering, 1999, pp. 256–267.
[23] L. Larsen, M.J. Harrold, Slicing object-oriented software, in: Proc. 18th International Conference on Software Engineering, 1996, pp. 495–505.

[24] J.K. Lee, S.J. Jung, S.D. Kim, W.H. Jang, D.H. Ham, Component identification method with coupling and cohesion, in: Proc. 8th Asia–Pacific Software Engineering Conference, 2001, pp. 79–86.
[25] E.L. Lehmann, Nonparametrics: Statistical Methods Based on Ranks, Holden-Day, 1975.
[26] C. Lindig, G. Snelting, Assessing modular structure of legacy code based on mathematical concept analysis, in: Proc. 19th International Conference on Software Engineering, 1997, pp. 349–359.
[27] D.H. Lorenz, J. Vlissides, Designing components versus objects: a transformational approach, in: Proc. 23rd International Conference on Software Engineering, 2001, pp. 253–262.
[28] H.A. Muller, M.A. Orgun, S.R. Tilley, J.S. Uhl, A reverse engineering approach to subsystem structure identification, Journal of Software Maintenance: Research and Practice 5 (4) (1993) 181–204.
[29] W.F. Opdyke, Refactoring object-oriented frameworks, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.
[30] Regexp, http://jakarta.apache.org/regexp/.
[31] Scientific Toolworks, Inc.: Understand for Java, http://www.scitools.com/uj.html.
[32] G. Snelting, F. Tip, Understanding class hierarchies using concept analysis, ACM Transactions on Programming Languages and Systems 22 (3) (2000) 540–582.
[33] M.S. Song, H.T. Jung, Y.J. Yang, The analysis technique for extraction of EJB component from legacy system, in: Proc. 6th IASTED International Conference on Software Engineering and Applications, 2002, pp. 241–244.
[34] SourceForge.net, http://sourceforge.net/.
[35] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1999.
[36] R.R. Valasareddi, D.L. Carver, A graph-based object identification process for procedural programs, in: Proc. 5th Working Conference on Reverse Engineering, 1998, pp. 50–58.
[37] H. Washizaki, H. Yamamoto, Y. Fukazawa, A metrics suite for measuring reusability of software components, in: Proc. 9th IEEE International Symposium on Software Metrics, 2003, pp. 211–223.