A QUICK SUMMARY : REFACTORING CASE

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

Component-based software development (CBD) didasarkan pada pembangunan perangkat lunak dari komponen perangkat lunak yang sudah ada sebelumnya. Dalam cbd, penggunaan kembali komponen umum dalam bentuk komponen dapat mengurangi biaya pengembangan system baru, dan mengurangi biaya pemeliharaan yang terkait dengan system ini. Namun biasanya program yang dibangun dengan mengunakan paradigma yang berorientasi object (OO) tidak dapat digunakan kembali dengan cepat dan efektif. In this paper, author mengusulkan teknik untuk mengekstraksi komponen dari program OO yang ada dengan menggunakan refactoring baru “Extract Component”. Teknik refactoring ini dapat mengidentifikasi dan mengekstrak komponen yang dapat digunakan kembali yang terdiri dari kelas-kelas program OO yang ada. Sebagai hasil dari evaluasi yang ada, ditemukan bahwa system ini dapat mengekstraksi reusable components dengan contoh pemakaian dari program java.

**INTRODUCTION**

CBD telah diterima secara luas sebagai pendekatan yang hemat biaya untuk pengembangan suatu perangkat lunak. CBD itu sendiri tidak harus selalu berorientasi objek; namun telah ditunjukkan bahwa menggunakan paradigma/ Bahasa OO adalah cara yang alami untuk memodelkan dan mengimplementasikan komponen. Dalam CBD, penggunaan kembali komponen umum dapat mengurangi biaya pengembangan system baru, dan mengurangi biaya pemeliharaan yang terkait dengan dukungan system ini. Namun, tidak semua komponen terkait dengan persyawaran fungsional sudah tersedia dalam semua konteks yang memungkinkan. Jika bagian dari program OO yang ada dapat dengan mudah digunakan kembali sebagai komponen, pemrogram dapat mengembangkan perangkat lunak dengan menggunakan CBD dengan memanfaatkan program ini. Tetapi biasanya kelas OO memiliki ketergantungan yang kompleks dan sulit untuk menggunakan kembali bagian dari program OO yang ada. Jika fungsi signifikan di realisasikan oleh satu set kelas, programmer yang ingin menggunakan kembali fungsi tersebut harus memeriksa dependensi di antara kelas terkait dan memperoleh semua kelas yang bergantung (dependen). Karena kegiatan pemeriksaan manual semacam ini memerlukan biaya tinggi bagi pemogram, manfaat reusable itu dapat itu dapat dikurangi atau dihilangkan. Oleh karena itu, pelu untuk mengubah bagian dari program OO yang ada menjadi komponen yang tidak memiliki ketergantungan pada elemen diluar itu sendiri.

Selain itu, bahkan jika komponen dapat diekstraksi dari program yang sudah ada, sulit untuk mengidentifikasi pengunaan yang tepat dari komponen yang diekstraksi hanya dengan merujuk pada source code atau interface public dari komponen. In this paper, mengusulkan teknik untuk mengidentifikasi bagian kandidat yang dapat digunakan kembali secara structural dari progam OO dan mengubah bagian ini menjadi komponen yang reusable secara otomatis oleh refactoring baru kami “extract component”. Pertama-tama kita mendefinisikan grafik hubungan kelas yang mewakili hubungan antara kelas/antarmuka dalam program java target. Selanjutnya, menggunakan CRG, penulis mengusulkan teknik untuk mengekstraksi komponen dari program OO, dan mengubah bagian di sekitar komponen yang diekstraksi untuk memungkinkan bagian2 di sekitarnya menggunakan komponen yang baru di ekstraksi.

**2. Class relation graph**

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 Γ = (*V* , Λ, *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* ∪ *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*∪*EI*∪*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*⊆*VC*×*V* ∪*VI*×*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 −◃.

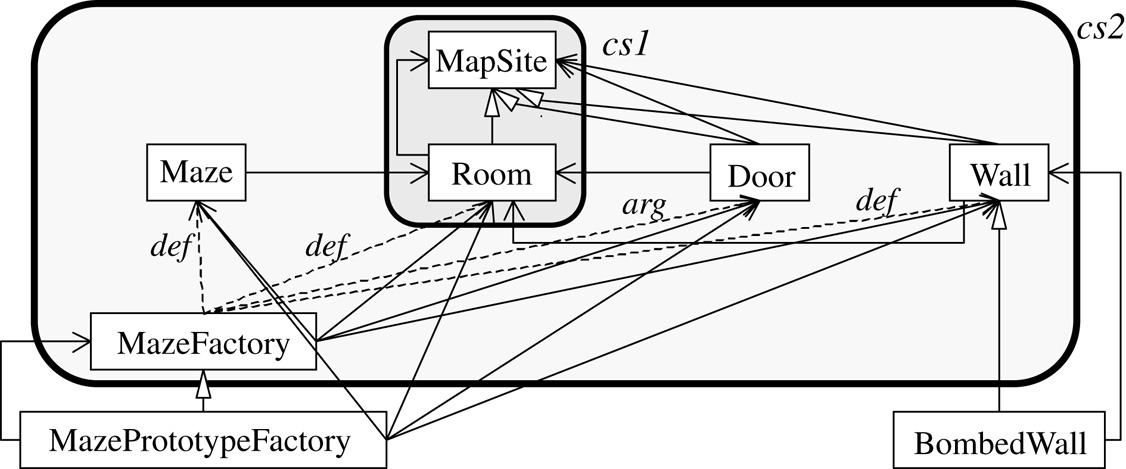
4. *EI* = *EID* ∪ *EIA*

*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* ⊆ *V* × *VC* × Λ

*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

arguments) of the target class for instantiation, and is denoted as · · ·>.



**Fig. 1. CRG of prototype pattern’s code.**

6. *EIA* ⊆ *V* × *VC* × Λ

*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 · · ·>. 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 anoter class *c*′ by using the *c*′’s constructor whose *def* ′

type of argument is int, we denote this as *Node*(*c*) · · ·> *Node*(*c* ).

7. *ER* ⊆ *V* × *V* × Λ

*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 →. In this paper, we (and our extraction system described in Section 5.1) recognize that a class/interface *ca* refers to another class/interface *cb*, when there is a type specification of *cb* for the variable/field/method declarations in *ca*, or there is a type specification of *cb* for the type casts in *ca* , or *ca* accesses a method/field of *cb* . 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:

*VC* = { BombedWall, Door, Maze, MazeFactory, MazePrototypeFactory, Room, Wall }

*VI* = { MapSite }  
*EE* = { (BombedWall,Wall), (Door,MapSite), (Room,MapSite), (Wall,MapSite), (MazePrototypeFactory,MazeFactory) }

*EID* = { (MazeFactory,Maze,9), (MazeFactory,Wall,10), (MazeFactory,Room,11) }

**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* = { (MazeFactory,Door,12) }  
*ER* = { (BombedWall,Wall,0), (BombedWall,Room,1), (Door,MapSite,2), . . . }.

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.