On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms

Article *in* ACM SIGSOFT Software Engineering Notes · August 2008

DOI: 10.1145/1402521.1402527 · Source: DBLP

CITATIONS
5

READS
4,205

3 authors:

Vinita Jindal
Keshav Mahavidyalaya, University of Delhi, Delhi, India
45 PUBLICATIONS **426** CITATIONS
SEE PROFILE

Amita Jain
Ambedkar Institute of Advanced Communication Technologies and Research
64 PUBLICATIONS **520** CITATIONS
SEE PROFILE

Devendra Kumar Tayal
IGDTUW
57 PUBLICATIONS **567** CITATIONS
SEE PROFILE

On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms (May 2007)

Vinita, Amita Jain, Devendra K. Tayal

***Abstract*—Reverse engineering is the key idea for reconstruction of any existing system. In this paper, we propose an algorithm to reverse engineer an object-oriented code into Unified Modeling Language (UML) class diagram. Our algorithm is very general in nature and can be applied to any object-oriented code irrespective of the object-oriented programming language. In our paper we consider an object-oriented pseudocode similar to C++ to implement our algorithm. Some of the researchers have dealt in the past the problem of reverse engineering an object-oriented code to UML class diagrams. However, none of these researchers have treated all the constructs available in UML class diagrams. Unlike the previously done work on reverse engineering into UML, our algorithm generates rules for a complete set of constructs available in UML class diagrams. It includes classes, relationships, objects, attributes, operations, inheritance, associations, interfaces & other extensible mechanisms also. This algorithm can be viewed as a solution to reverse engineer any available object-oriented software. An application for the implementation of above said rules using C++ code is also included in the paper. We thoroughly compare our work with the similar type of earlier work in this area and uncover the deficiencies in these previous available works. Moreover our motive in this paper is to prepare rules to reverse engineer C++ code into UML class diagrams and not to generate any tools.**

***Index Terms*—Class diagrams, Object-Oriented Programming, Reverse engineering, UML.**

I. Introduction

Reconstruction plays an important role in increasing the quality and productivity of software development. It involves activities for abstracting information that ranges from source-level information to higher-level views of that information. Reverse engineering is the key idea for reconstruction of any existing system [1]. For this, we require that the deliverable produced should satisfy the evolving needs of its users.

It is a four-stage process that includes:

- Analysis of product
- Generation of an intermediate level product description
- Human analysis of product description to produce a specification, and
- Generation of new product using specification

Production of such software requires the construction of models to visualize and control the systems evolving requirements and structure. For this purpose, we use the UML because of its expressiveness & completeness. UML is the *de facto* standard for visualization, specification, construction and documentation for a system. In particular the class diagrams supported by UML visualize and model the static aspects of the system under development in a complete and unambiguous way [2], [3]. These capabilities make UML as the most appropriate language for reverse engineering of a system. When we reverse engineer object-oriented code into class diagram, the relationships between classes are determined according to the type of instance variables and methods parameters. There is no standard bridge between C++ and UML, and all reverse engineering applications tend to build this [4].

We first present a brief survey of the previous work done in this area. In [5] Sutton *et al.* propose the similar approach like ours. But they implement their concepts in a tool "pilfer" to reverse engineer C++ code into UML class diagrams. However, the object-oriented concepts used by them were defined using object-oriented language rules as in C++, Java etc. So there can be some conflict/variation between the definition and implementation of a concept in different object-oriented languages. Their work is therefore language dependent. In our paper we use the object-oriented concepts as defined in UML, which are unanimously accepted. Moreover we also discuss the visibility of attributes & operations, abstract classes, template classes, structures, packages, comments and extensible mechanisms etc. that are not dealt in [5]. Tonella *et al.* [6] propose a flow analysis algorithm with emphasis on reverse engineering of typed information by identifying weakly type containers, however they do not deal with the reverse engineering of relationships, objects, extensible mechanisms etc. which are explicitly dealt by us in this paper. In [7] Guéhéneuc *et al.* discuss the recovering of only binary class

Manuscript received May 23, 2007. For paper title "On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms".

Vinita is with the Keshav Mahavidyalaya, Delhi University, Delhi, India (phone: 0-98101-00377; e-mail: vinita13jindal@gmail.com).

Amita Jain. is with Guru Premsukh Memorial College of Engineering, GGSIPU, Delhi, India (e-mail: amita_jain_17@yahoo.com).

Dr. Devendra K. Tayal is with Jaypee Institute of Information Technology University, Noida, U.P., India (e-mail: devendra.tayal@jiit.ac.in).

relationship from an object-oriented code by considering the various properties of relationship. They have not proposed any criteria for reverse engineering of other associated constructs. Jackson *et al.* [8] describe a tool "womble" for automatic extraction of object model from given Java code with a complete focus on multiplicity only. Although their paper provides a description of the tool, nothing is said about the rules that are being used for reverse engineering. In our paper we are proposing the rules for reverse engineering of object-oriented code into UML class diagrams rather than implementation of any tool. Kollmann *et al.* [9] discuss an approach to discover only the association patterns in the code and represent those using advanced UML adornments in class diagrams using Java language only. However they miss to provide complete procedure to reverse engineer a full object-oriented code. On the other side our approach is more general & complete as it also includes basic constructs of object-oriented code such as classes, objects, structures etc. Matzko *et al.* [10] takes an approach similar to ours – the implementation for reverse engineering a C++ code. However, their work focuses on the modeling of C++ syntax in UML rather than attempting to recover any design abstractions involved. Moreover, very little is said about rules for multiplicity or aggregation semantics, which on the other side is supplied by our paper.

As compared to the previous approaches our algorithm can be viewed as a solution for reverse engineering any available object-oriented code. This algorithm includes all possible constructs available in UML. An application for the implementation of above said rules is also included in the paper.

The paper is organized as follows: Section I covers introduction. Section II proposes an algorithm to reverse engineer object-oriented code into UML class diagram incorporating extensible mechanisms. Section III demonstrates the application of the approach to a code given in C++. In the last section, conclusions are derived.

II. Algorithm For Reverse Engineering

In this section we devise an algorithm to reverse engineer an object-oriented code into UML class diagrams. The input of the algorithm is an object-oriented code and the output is corresponding UML class diagram. Our algorithm is divided into ten major steps to deal with things (classes), objects, relationships and extensible mechanisms.

ALGORITHM

INPUT: An object-oriented code

STEP 1: (IDENTIFY STRUCTURAL THINGS): The nouns of UML models are called structural things. These are mainly divided into class, abstract class, template class and interface. For identifying structural things, we define the following rules: -

Rule I: Identify the keyword ***class*** in the code and represent it in UML as Fig. 1.1

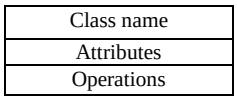


Fig. 1.1

Make a class in UML such that the *class names* in UML diagram is given as the same name as that of code class name. *Attributes* are the variables of code class and will be represented as

[Visibility] attribute_name [multiplicity][: data_ type][=initial_value]

Operations are the function of code class and will be represented as

[visibility] function_name [(parameter_list)][: return_ data_ type]

(i) Visibility of attribute and operations is identified using Fig. 1.2

Keywords	Representation
Private	-pvt_var_name
Public	+pub_var_name
Protected	#prot_var_name
Derived	/der_var_name
Abstract	*abst_var_name*
Static	$stat_var_name
Package	~package_name

Fig. 1.2

In case no keyword is present for attributes and operations, they will be represented as *private.*

(ii) Multiplicity is the number of instances of a class. It can be identified in an object-oriented code as given in the Fig. 1.3

Class	Multiplicity
Single instance class	1
Abstract class	0
Basic default class	*
Specific number of instances (n)	n

Fig. 1.3

(iii) Parameter_list is defined as

parameter name: data type [=default_value]

Consider a code as given in Fig. 1.4. In this code we have a class Stack with two private attributes stck & tos and three public operations init, push & pop. According to **Rule I** stck is a private variable and tos is a private and static variable (Rule I (i)). Similarly init, push & pop are public operations (Rule I (i)). Thus the UML class diagram corresponding to Fig. 1.4 is shown in Fig. 1.5.

```
Class Stack {
int stck;
static int tos;
public:
void init ();
void push (int i);
int pop ();};
```

Fig. 1.4 Class in C++ Language

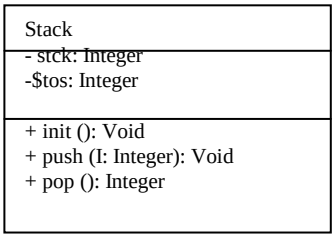


Fig. 1.5 UML class diagram corr. to Fig. 1.4

Matzko *et al.* [10] and Sutton *et al.* [5] also identify classes in their papers. However, they both have not specified any mapping rules. Also the visibility of attributes and operations is not being discussed by them. Jackson *et al.* [8] and Kollman *et al.* have discussed the concept of multiplicity in their paper [9] but no rule is being specified. In our paper **Rule I(i), I(ii) & I(iii)** of the algorithm specify the mapping to identify the visibility, multiplicity and parameter_list along with a class.

Rule II: Abstract class can be identified by using the following rules:

- It is either followed by keyword *abstract*, or
- It is a class without instance, or
- It contains at least one pure virtual function.

It is represented in UML in exactly same way as a class with the exception that name of classes, attributes and operations will be written in Italics.

Consider a code as given in Fig. 2.1. The corresponding UML diagram according to **Rule II** of algorithm is given in Fig. 2.2.

```
Class Item
{
public:
 virtual void cut()=0;
 virtual void move (int x,
    int y) = 0;
 virtual Boolean pick (int x,
    int y) = 0;
 virtual void ungroup ()
  = 0; };
```

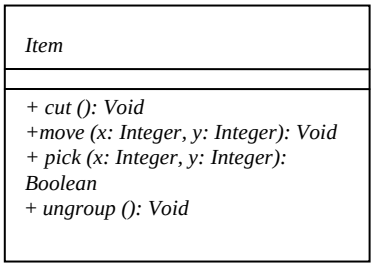


Fig 2.1 Abstract class in C++

Fig 2.2 UML diagram corr. to Fig 2.1

None of the researchers in the past have proposed a procedure to identify the abstract classes in the object-oriented code although abstract classes are mainly used in an object-oriented code to implement interfaces. Also these are important to keep a program organized and understandable. Abstract classes and methods reveal design intentions of the designer. The **Rule II** explicitly specifies the mapping to an abstract class in corresponding UML class diagram.

Rule III: Template classes can be mapped by the following rules:

- It is either followed by keyword *template* in the code, then it will be represented in UML as in Fig. 3.1, or
- It is a class instantiated from template in given code, then it will be represented in UML as in Fig. 3.2

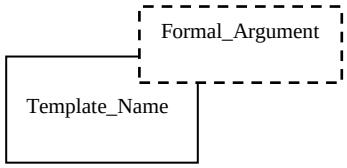


Fig. 3.1

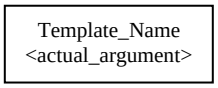
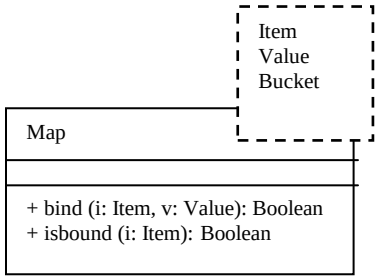


Fig. 3.2

Consider a code as given in Fig. 3.3. The corresponding UML diagram according to **Rule III** of algorithm is given in Fig. 3.4.

```
Template <class Item, class Value, int Bucket >
Class Map{          (a)
    public:
    boolean bind (Item I,
        Value v);
    boolean isbound (Item
          i);
          ……
          ……
          ……
    };
(a)  m: Map <customer,
     order, 3>;
```



(b)

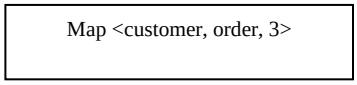


Fig. 3.3 Template class and its instance

Fig. 3.4 UML diagram corr. to Fig. 3.3

In an object-oriented code template classes are used where we have multiple copies of code for different data types with the same logic. Thus, templates are used to implement an efficient code base that is reusable and extensible. In our paper, **Rule III** proposes the rules for identifying and mapping template classes occurring in an object-oriented code into UML class diagrams.

Rule IV: Identify the keyword ***struct*** in the code. Represent it in UML as a class (as in Fig. 1.1). The name of the UML class will be same as the structure name. The attributes and operations of structure will be declared **pubic** by default, if no keyword is present explicitly.

Consider a code as given in Fig. 4.1. The corresponding UML diagram according to **Rule IV** of algorithm is given in Fig. 4.2.

```
Struct node
{
        int data;
        char name;
};
```

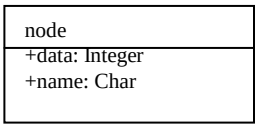


Fig. 4.1 Structure in C++ language

Fig. 4.2 UML class diagram corr. to Fig. 4.1

None of the earlier researchers proposed any mappings for structures. Thus the importance of structure in any object-oriented code is left uncovered. Although whenever one wants to declare class with mainly public members, it is recommended to use structure in place of classes for better understanding. So in our paper we specify the **Rule IV** for mapping for structures in corresponding UML class diagrams.

STEP 2: (IDENTIFY BEHAVIORAL THINGS): The dynamic parts of UML models are called behavioral things. These are mainly having interaction that comprises of a set of messages exchanged among objects to accomplish a specific purpose. Thus the behavior of a class or an object may be specified with interaction. Interaction involves messages, action sequences and links. For identifying the behavioral thing, we define the following rule: -

Rule V: We identify the **interaction** within function parameters or declaration of classes and/or functions with user consultancy,

in case they are not present in object-oriented code. Generally interaction shows the behavior that comprises a set of messages exchanged among a set of objects. **Interaction includes the name of its operation.** Interactions will be represented in UML by a directed line as in Fig. 5.1.

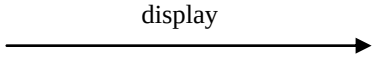


Fig. 5.1 Messages

Since behavioral things are the dynamic part of UML model, we are not discussing these in detail. So in our paper we are just specifying its representation in UML class diagrams.

STEP 3: (IDENTIFY GROUPING THINGS): The organizational parts of UML models are called grouping things. These are boxes into which model can be decomposed. A "Package" is only primary kind of grouping thing. For identifying the package, we define the following rule: -

Rule VI: Identify the keyword **package** or **headers files** with .h extension in the code. Then, represent them in UML as in Fig. 6.1. In case keyword **namespace** is present in the code, then represent it in UML as in Fig. 6.2 with all classes included inside the rectangle.

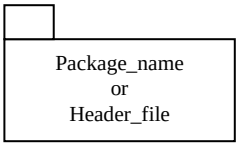


Fig. 6.1

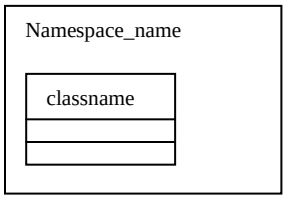


Fig. 6.2

Consider a code as given in Fig. 6.3. The corresponding UML diagram according to **Rule VI** of algorithm is given in Fig. 6.4.

```
# include<iostream.h>
class Helloworld {
 public:
void paint ()
{
  count << " HELLO
        WORLD";}
};
```

Fig. 6.3 Packages in C++ language

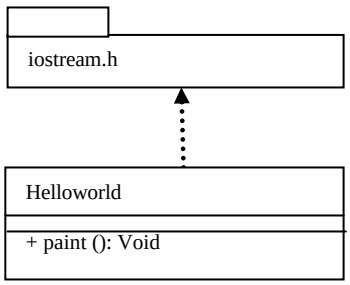


Fig. 6.4 UML diagram corr. to Fig. 6.3

In an object-oriented code we use packages, header files or namespaces for organizing elements into groups. They simplify the maintenance and reusability of object-oriented code. Packages exhibit functionally cohesive services. Namespace improves the structure of object-oriented codes. Header files are needed for linking a program. This makes reverse engineering of packages, header files and namespaces as an essential component. So, **Rule VI** specifies the mapping for packages, header files and namespaces in corresponding UML class diagrams.

STEP 4: (IDENTIFY ANNOTATIONAL THINGS): The explanation parts of UML models are called annotational things. "Note" is only primary kind of annotational thing. Note is used to adorn diagrams with constraints or comments that are best expressed in informal or formal text. The mapping for reverse engineering of notes is given by the following rule: -

Rule VII: Identify **comments** in code, they are identified in code as //……. Or /*……..*/ and represent them as **note** to the class in UML as in fig. 7.1

Fig. 7.1

Consider a code as given in Fig. 7.2. Its corresponding UML diagram according to **Rule VII** of algorithm is given in Fig. 7.3

```
// Class Declaration For Person

class Person
{
private:
        char name;
        int age;
};
```

Fig 7.2 Comments

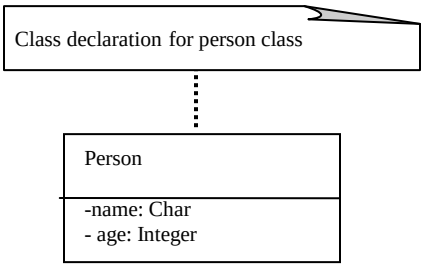


Fig. 7.3 UML diagram corr. to Fig. 7.2

Comments are used to describe, illuminate and remark about any element in a model. Benefits of using comments are better understanding and their flexibility. Hence reverse engineering of comments must also be present in the design document for better understanding of the design. In our paper, **Rule VII** explicitly specifies the mapping for comments in corresponding UML class diagrams.

STEP 5: (IDENTIFY OBJECTS): To identify the object from any object-oriented code, we define the following rules: -

Rule VIII: Identify the **instances of the class (objects)** in the code, which have a class as their data type. Represent them, as classes (as in Fig 1.1) with the difference that object name will be underline in the UML representation. Also identify all **stand-alone functions** in code and represent them as dotted rectangle in UML.

Consider a code as given in Fig. 8.1. The corresponding UML diagram according to **Rule VIII** of algorithm is given in Fig.8.2.

```
Class Stack {int stck;
static int tos;
public:
void init ();
void push (int i);
int pop ();
};
int main (){
        Stack a, b;
}
```

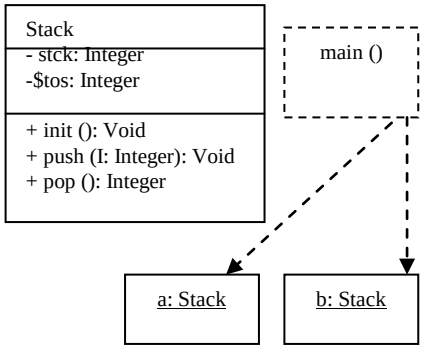


Fig. 8.1 Object in C++ Language

Fig. 8.2 UML class diagram corr. to Fig. 8.1

In any object-oriented code the whole processing is done through objects only, so they are the major part of code. Objects provide flexibility and a higher level of abstraction in object-oriented programming environment. Thus, Reverse engineering of objects will play a significant role in understanding the design document. Their role becomes more important in forward engineering of the object-oriented design. Hence our **Rule VIII** specifies the corresponding mapping for objects in UML class diagram representation.

STEP 6: (IDENTIFY DEPENDENCY): Dependency is a kind of relationship, which states that a change in specification in one class may affect another class that uses it.

Rule IX: In context of classes, the **dependency** relationship can be identified when one class uses another class as an argument in the signature of an operation and if used class changes, the operation of other class may be affected as well. Dependency is represented in UML by dotted line with arrows between the classes.

Consider a code as given in Fig. 9.1. Here, we have two classes as Filmclip and Channel. They will be represented in UML class diagram according to Rule I. Again, a public function 'playon' of class Filmclip is using the object 'c' of class Channel in its signature as shown in Fig 9.1. So, it gives rise to "dependency". The corresponding UML diagram according to **Rule IX** of algorithm is shown in Fig. 9.2.

```
class Channel {public:
        float band;};
class Filmclip {public:
   char * name;
   void playon (Channel c);
    void start ();
    void stop ();
    void reset ();};
```

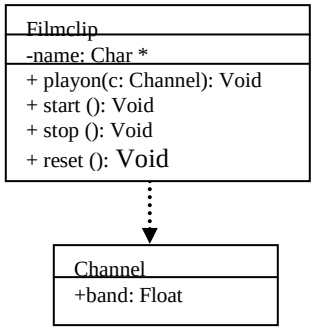


Fig. 9.1 Classes in C++

Fig. 9.2 UML diagram language corr. to Fig. 9.1

The dependency is not explicitly shown in code, so none of the earlier researchers might have felt the need to discuss this concept. But dependency plays an important role in the understanding of code as the relationship correctly depicts that how the effects of a change in one class is being propagated into other class. During the process of reverse engineering it is necessary to incorporate dependency not to lose any important information regarding lower level abstraction and/or any other functional or structural representation. So our **Rule IX** explicitly specifies the corresponding UML class diagram representation.

STEP 7: (IDENTIFY ASSOCIATION): Association is a type of HAS_A relationship for **whole/part** relations. Association specifies that objects of one class are connected to the objects of another class.

Rule X: (Discover Association): Association can be identified by considering the attributes in code when there are two classes A and B such that A has an object of class B and B has an object of class A. In UML, we will be represent it by making a solid line between the two classes A and B as shown in Fig. 10.1

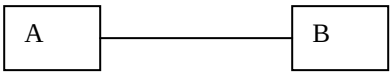


Fig. 10.1

The association relationship also supports several adornments like: (i) name, (ii) role name, and (iii) multiplicity.

However there is no provision to identify these directly from code. Hence these are entirely dependent on the perspective of the designer. So, the designer may choose them appropriately after taking in view the semantic of classes.

Consider a code as given in Fig. 10.2.We have two classes Company and Person. Here, the attribute of Person class is of Company type and that of Company class is of Person type. Thus, two classes are having association relationship between them. The corresponding UML diagram according to **Rule X** of algorithm is shown in Fig. 10.3.

```
class Company
{private:
 Person employee[];
 /*...*/ }
class Person
{private:
 Company employer;
  / * . . . * / }
```

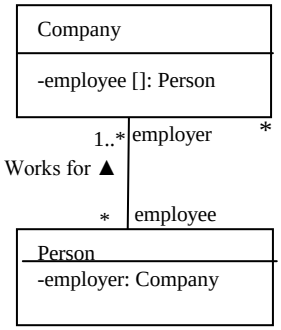


Fig. 10.2 Association in C++ language

Fig. 10.3 UML diagram showing name, role name and multiplicity of Association corr. to Fig. 10.2

The association relationship is used as a measure of closeness among the system files. Also it can be used for finding data sharing. So, it is very important to include reverse engineering of association in the algorithm. But as association cannot be

obtained directly from code, perhaps nowhere rule is being specified for association, which is given by **Rule X** of our paper.

Rule XI: (Discover aggregation): Aggregation is a special form of association that models **whole/part** relationship. **Aggregation** can be identified by considering two classes A and B such that **lifetime of A and B are independent** and class B contains an object of class A, then class B is the aggregation of class A. In UML we will represent it as shown in Fig. 11.1.

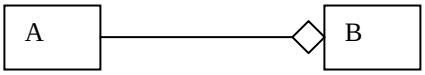


Fig. 11.1

Consider a code as given in Fig. 11.2. Here we have two classes as Dept and Company. Here dname, an attribute of Company is of Dept type. Also the instance of Company is not dependent on Dept class, i.e. their lifetimes are independent. Hence Company class is the aggregation of Dept class. The corresponding UML class diagram according to **Rule XI** of algorithm is shown in Fig. 11.3.

```
class Dept
{
 /*...*/
}
class Company
{private:
 Dept dname;
  /* . . . */ }
```

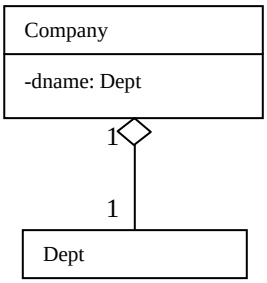


Fig. 11.2 Aggregation in C++ language

Fig. 11.3 UML diagram corr. to Fig. 11.2

In [9] Kollmann *et al.* are using similar approach like ours with the difference that they are using only Java language for reverse engineering while our rule can be implemented for any object-oriented programming language. In [7] Guéhéneuc *et al.* find the aggregation based only on whole/part instance of class but in our paper we also incorporate the lifetime of classes.

Here, we consider the fact that the aggregation implementation is based on the idea " The instance of aggregated class can be shared by many other classes. While those classes are deleted, the instance of aggregated class won't be deleted". Thus, the aggregation relationship is very important in many domains and should be manifested whenever possible. Aggregations can guide programmers in their implementation work as well as the understanding of the system.

For instance in Fig. 11.3, Dept and Company are two different classes. In Company class, dname is an instance of Dept class .Now, if we delete the instance dname of class Dept from Company, as it will independent of Dept class. Hence Company is the aggregation of Dept.

Rule XII: (Discover composition): Composition is another special form of aggregation. **Composition** can be is identified by considering two classes A and B such that **B control the lifetime of A** and B contains an object of class A, then B is the composition of A. It will be represented in UML as shown in Fig. 12.1.

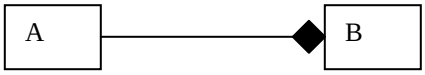


Fig. 12.1

Consider a code as given in Fig. 12.2 below. Here we have two classes Point and Circle. Here center, an attribute of Circle is of Point type and instance of Circle is dependent on Point class, i.e. their lifetime is dependent. Hence Circle is the aggregation of Point. The corresponding UML diagram according to **Rule XII** of algorithm is shown in Fig. 12.3.

```
class Point
{
 /*...*/
}
class Circle
{private:
 Point center;
  /* . . . */ }
```

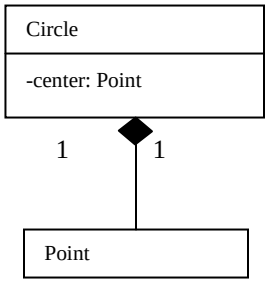


Fig. 12.2 Composition in C++ language

Fig. 12.3 UML diagram corr. to Fig. 12.2

In [9] Kollmann *et al.* propose the similar rules for composition. However their work is confined only to Java language. While rules generated by our work can be applied to any object-oriented programming language.

Here, we consider the fact that the composite implementation is based on the idea " the instance of composited class can only be used by one other class. While that class is deleted. The composited class will also be deleted". Other class beside the class that contains it can use composition class. However, the lifetime of the composite class cannot exceed the lifetime of the container class. Thus, composite relationship should also be manifested whenever possible.

For instance in Fig. 12.3, Point and Circle are two different classes. In Circle class, center is an instance of Point class and if we delete the instance of center from Circle, as it will depend on Point class, and there is no meaning of circle without Point. So, Circle class is dependent on Point class. Hence Circle is the composition of Point.

STEP 8: (IDENTIFY GENERALIZATION): Generalization is a type of IS_A relationship where all the derived classes are specialization of base class and all the *base classes* are generalization of derived classes.

Rule XIII: (Discover generalization/ inheritance) In context of classes it can be identified in the definition of class in code as it incorporates another class name into its declaration. It is written as, first we write the name of derived class, then use symbol ':' and finally write the name(s) of parent (base) class(s). In object-oriented code, first base class is defined, and then

derived classes are defined. **Visibility of derived classes** can be derived using the table given in Fig.13.

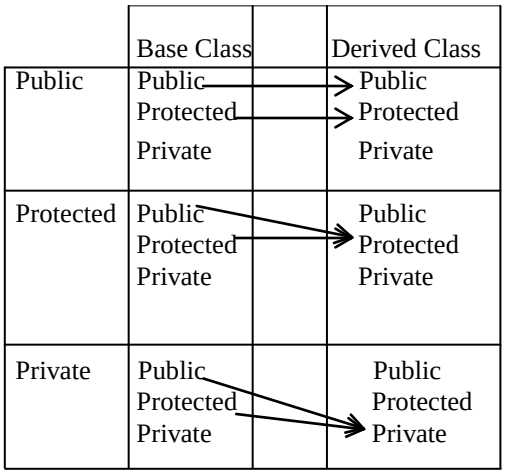


Fig. 13 visibility of derived classes

One of the major activities in software development is the process of generalizing components. We like generalized (inherited) code because it can be reused in various applications. When it has been thoroughly tested and its use properly documented, generalized code saves the user of that code considerable time and effort. Inheritance can be either simple inheritance or multiple inheritance depending upon the number of parents exist.

(i) A class that has exactly one parent uses ***simple inheritance*** shown in the code given in Fig. 13.1 [11]. The corresponding UML diagram is shown in Fig. 13.2.

Consider a code as given in Fig. 13.1. Here, we have three classes Customer, Human_customer and Inst_customer. These will be represented in UML using Rule I of the algorithm.Now, we find that initially base class Customer is declared, and then derived classes Human_customer and Inst_customer are being declared. Both of these derived classes are derived publicly, so members of base class are visible in derived classes according to Fig.13. Generalization is shown in UML using a symbol for generalization-specialization (a hollow-headed arrow) according to **Rule XIII** of algorithm. Final UML diagram representation is shown in Fig. 13.2.

```
class Customer {
protected: int custno;};
class Human_customer:
     public Customer {
 protected:
        char  sex;
        char bdate;};
class Inst_customer:
    public Customer {
 protected:
        int noofemp;
        char orgtype;};
```

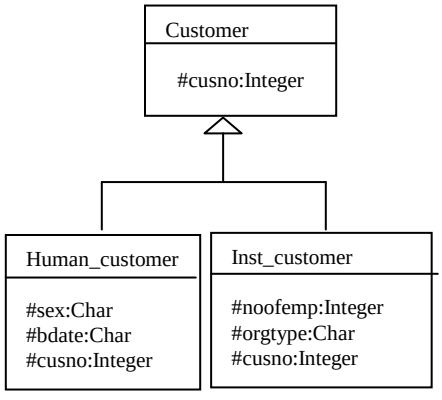


Fig. 13.1 Generalization (Simple inheritance) **Fig.13.2UMLdiagram corresponding to Fig. 13.1**

(ii) In case of more than one parent exist, that class uses **multiple inheritance** shown in the code given in Fig. 13.3 [11]. The corresponding UML diagram is shown in Fig. 13.4.

Consider a code as given in Fig. 13.3. Here we are given two base classes employee, mgt_traineee and one publicly derived class sal_mgt_trainee in the given code. The derived class is derived publicly, so members of both base classes are visible in derived classes using Fig. 13. Further private members in base classes will not be inherited in derived class. The corresponding UML diagram according to **Rule XIII** of algorithm is represented in Fig. 13.4.

```
class employee {
        char ssno;
        char * name;};
class mgt_traineee {
        char gender;
        char * prgm;};
class sal_mgt_trainee: public
 employee, public mgt_trainee
{  long salary;};
```

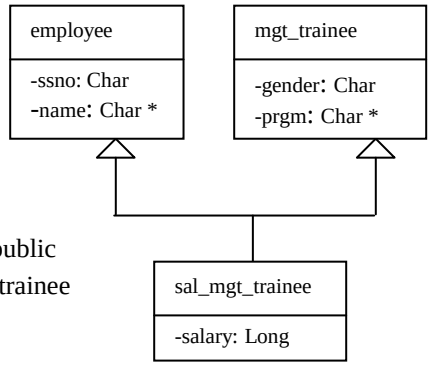


Fig. 13.3 Generalization (Multiple inheritance) **Fig. 13.4 UML diagram corresponding to Fig. 13.3**

In any object-oriented language code all the attributes and functions are generally shared using inheritance. It can be simple or multiple, but both can be identified by a careful examination only. Unfortunately earlier researchers did not explicitly discuss this important concept also .In our paper we devise the **Rule XIII** for reverse engineering of the same into UML class diagram. Here we also **devise Fig. 13**, to describe the visibility of derived attributes/ operations for derived classes.

STEP 9: (IDENTIFY REALIZATION): In a realization relationship, one entity (normally an interface) defines a set of functionalities as a contract and the other entity (normally a class) "realizes" the contract by implementing the functionality defined in the contract. In the UML, a realization relationship exists between two classes when one of them must realize, or implement, the behavior specified by the other. The class that specifies the behavior is called the supplier, and the class that implements the behavior is called the consumer. A realization relationship can include those between interfaces and classes. The interface specifies the behaviors, and the subsystem implements the behaviors.

Rule XIV: (Discover realization or interface): Realization can be mapped by the following rules:

- It is either followed by keyword ***interface*** in the code, or
- It may be any class in the code that defines only public, pure virtual functions with no member variable

declaration. In case, it is derived, corresponding base classes must also be interfaces.

It will be represented in UML either as class with stereotype <<Interface >> or simply as an interface with circle attached with implementing class.

Consider a code given in Fig. 14.1. Here name of interface is same as the structure name i.e. linklist. Further, by default all the members of structure are public by nature so their visibility will be denoted by + and class list is using this interface. The corresponding UML diagram according to **Rule XIV** of the algorithm is represented in Fig. 14.2 or simply as an interface with implementing class is represented in Fig 14.3.

```
struct linklist {
 virtual int add (int *)=0;
 virtual void remove (
    int *)=0;
 virtual boolean contains
    (int *)=0;
 virtual void clear ()=0;
} ilist;
class List:: ilist
{public:
      struct linklst l;}
```

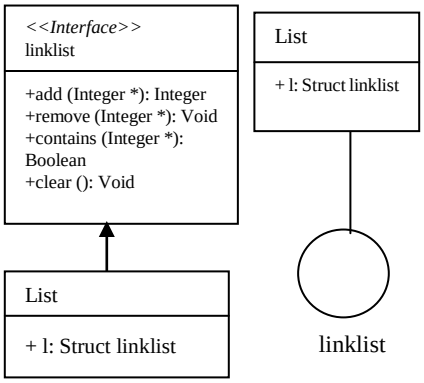


Fig. 14.1 Interface in C++

Fig. 14.2 UML class diag. for to Fig. 14.1

Fig. 14.3 UML class diag. for Fig. 14.1

The main use of realization or interface is reusability. It allows us to reuse a lot of code and make things simpler at the same time. Interfaces promote abstraction. In case we want to use some class for its functions only, we have to define the interface in which we just use implementation part only. So, **Rule XIV** of the algorithm specifies the corresponding alternate UML class diagram representations.

STEP 10: (IDENTIFY EXTENSIBLE MECHANISMS): The extensible mechanisms available in UML permit the extension of the object-oriented languages in controlled manner. To identify the extensible mechanisms from any object-oriented code, we define the following rules: -

Rule XV: The semantic of **extensible mechanisms** depends on a convention of the user or an interpretation by a particular constraint language or programming language. These include *stereotype, tagged value, and constraints*.

- A ***stereotype*** denotes a variation on an existing modeling element with the same form but with a modified intent. Stereotypes are effectively used to extend the UML in a consistent manner and denoted within guillemets (<< >>).

Very few researchers have this information explicitly documented in their paper. We feel method stereotype information forms the basis for supporting more sophisticated types of design recovery. Knowing class stereotypes allows us to determine architectural importance for automated layout of class diagrams or architectural level understanding. A stereotype is a high-level description of the role of a method. It gives a clear picture of what a method does and its responsibilities within the class. Stereotypes widely recognized by the development and maintenance communities include *constructor*, *destructor*, interface, template and exception etc. So, a rule is provided by our paper for reverse engineering of stereotype from code into UML class diagram.

- A ***tagged value*** is a keyword-value pair that may be attached to any kind of model element and denoted with a comma-delimited sequence of property specifications all inside a pair of braces ({}).

Tagged values extend model elements with new kinds of properties with the format of a pair of name and an associated value, i.e., {name=value}. Also, they allows us to create new information in the element's specifications. So our rule specifies the corresponding UML class diagram representation.

- A ***constraint*** is a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true and denoted with a text string in braces ({}). A constraint string following a list entry maps into a Constraint attached to the element corresponding to the list entry.

Constraints add new semantic restrictions to a model element, or modify existing ones. So our rule specifies the corresponding UML class diagram representation.

Consider a code given in Fig. 15.1. Here, we only want to throw and caught the exception for the class Overflow. Further we want to write the author and version of EventQueue class and want to add the constraint that all additions are done in order. The corresponding UML diagram according to **Rule XV** of the algorithm is represented in Fig. 15.2.

```
class EventQueue {

        add();
        remove();
         };
class Overflow{
      ----------
};
```

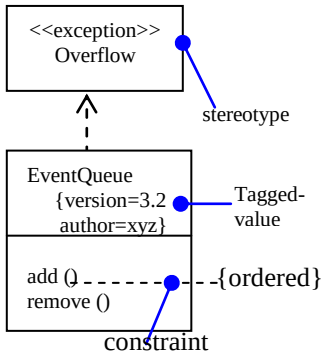


Fig. 15. Extensible Mechanisms

Fig. 15.2 UML diagram corresponding to Fig. 15.1

Extensible mechanisms permit the extension to understand any concept so that one knows what convention a particular user uses. The application of a design pattern may change the names of classes, operations, and attributes participating in this pattern to the terms of the application domain. Thus, the roles that the classes, operations, and attributes play in this pattern have lost. Without explicitly representing pattern-related information, the designers are force to communicate at the class and object level, instead of the pattern level. The design decisions and tradeoffs captured in the pattern are lost too. Our approach uses the UML extension mechanisms to define a UML profile for visualizing

design patterns. Thus, the notations provided in this paper help on the explicit representation of design patterns.

III. APPLICATION

In this section we now show how the class diagrams is constructed by using the algorithm proposed by our paper. We are taking a code in C++ to implement our algorithm. Consider a given C++ Code example given below in Fig. 16 This figure shows a complete C++ program including six classes, a namespace and function main**.**

```
namespace Security {
class Passwd {
        private:  char * pwd;
        public:
                Passwd ();
                Passwd (char * p);
                void setPasswd (char * p);};}
class Person {
        protected:
                char * nm;
        public:
                Person (char * nm);
                Person ();
                Char * getName ();
                virtual void print ()=0;  };
class Bankaccount;
void check ();
class Customer: public Person {
        private:
                static int len;
                Bankaccount * b;
        public:
                friend void check ();
                Customer (char * nm);
                static int getpasswdlen ();
                void print ();
                void checkPasswd ();  };
class Company {
        protected:
                float prft;
        public:
                Company (float pr);
                Company ();
                Float getProfit (); };
class Bankaccount {
        private:
                float tot;
                Customer hld;
        public:
                Bankaccount (Customer c);
                void deposit (float amt);
                float getTotal ();  };
class Bank: public Company {
        private:
                Bankaccount * actlst;
        public:
                Bank (float prft);
              void addAccount (const Bankaccount &b);  };
int main (){
                Customer mycust ("xyz");
                Bank b (1000);
                b.addAccount (Bankaccount (mycust));
                return 0;   }
```

Fig. 16 C++ Code Example. This figure shows a complete C++ program including six classes, a namespace and function main.

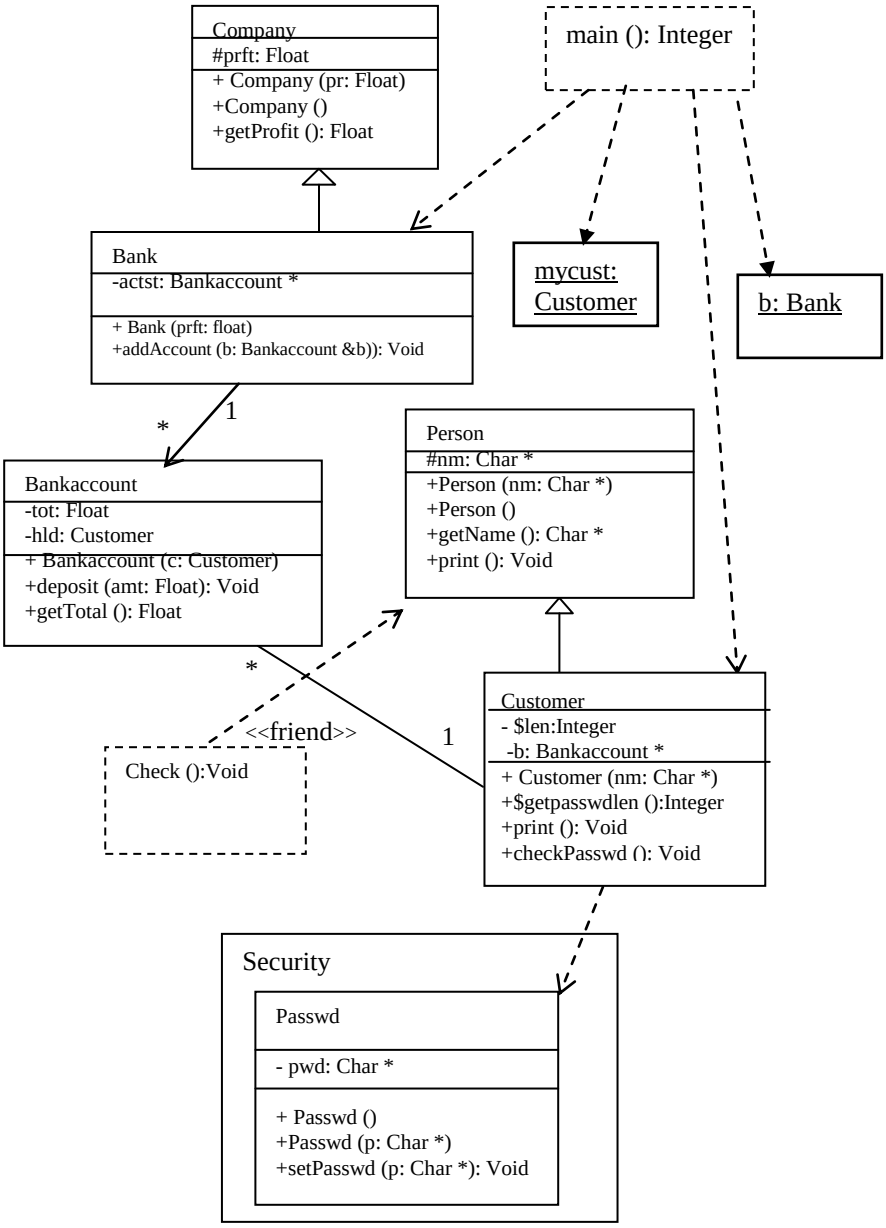


Fig. 17 corresponding to Fig. 16

In the above example we identify all the classes (**Company, Bank, Bankaccount, Person, Customer and Passwd)** in the code and represent them in UML according to Rule I. Now, we identify namespace **Security** in code and represent it in UML according to Rule VI. Further, we identify the objects **mycust**

and **b** in main function. Also we identify standalone functions **check()** and **main() a**nd represent them in UML according to Rule VII. Next, in class **Bankaccount** the public function Bankaccount is using the object of the **customer** class in its signature. It gives rise to dependency using Rule X. **Customer** class incorporates the name of **Person** class in its definition and **Bank** class incorporates the name of **Company** class in its definition. This gives rise to generalization using Rule XIII of the algorithm. After applying all rules, the corresponding UML diagram according to our algorithm is shown in Fig. 17.

IV. Conclusion

We have introduced an algorithm to reverse engineer an object-oriented code into UML class diagrams incorporating extensible mechanisms. The algorithm presented in this paper maps all constructs available in UML class diagrams and is independent of any object-oriented programming language. This algorithm maps classes, abstract classes, template classes, packages, header files, namespaces, comments, object diagrams, various relationships and extensible mechanisms into corresponding UML class diagrams. We have also proved our work by taking an application in C++ implementing the algorithm. We have compared our work with similar type of work in this area. Our algorithm may serve as a tool for reverse engineering any object-oriented code into UML class diagram which may be later on forward engineered for the development of any other software or program.

References

[1] Pressman, R.S. (2001), Software Engineering, A Practitioner's approach, 5th Ed., Mc Graw-Hill Series.

[2] Booch, G., Rambaugh, J. and Jacobson, I. (1999), The Unified Modeling Language User Guide, Object-Oriented Technology series, Addison-Wesley.

[3] Lee, R.C. and Tepfenhart, W.M. (2001), UML and C++, A practical guide to Object-Oriented development, 2nd Ed., Prentice-Hall.

[4] Schildt, H. (1998), C++, The Complete Reference, 3rd Ed., Mc Graw-Hill Series.

[5] Sutton, A. and Maletic, J.I. (2005), "Mapping for accurately Reverse Engineering UML class models from C++" in proceedings of 12th working conference on Reverse Engineering (WCRE'05), Nov 7-11 2005, page(s): 10 pp.

[6] Tonella, P. and Potrich, A. (2001), "Reverse Engineering UML class diagram from C++ code in the presence of weakly typed containers" in proceedings of International conference on Software Maintenance (ICSM'01), Florence, Italy, Nov 6-10 2001, pp: 376-385.

[7] Guéhéneuc, Y. -G. and Albin-Amiot, H. (2004), "Recovering Binary Class Relationships: Putting Icing on the UML Cake", in Proceedings of 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), Vancouver, Canada, Oct 24-28 2004, pp. 301-314.

[8] Jackson, D. and Waingold, A. (1999), "Lightweight Extraction of Object Models from Bytecode", in Proceedings of 21st International Conference on Software Engineering (ICSE'99), Los Angeles, California, May 16-22 1999, pp. 194-202.

[9] Kollman, R. and Gogolla, M. (2001), "Application of UML Associations and Their Adornments in Design Recovery", in Proceedings of Eight Working Conference on Reverse Engineering (WCRE'01), Stutgart, Oct 2-5 2001, pp. 81-92.

[10] Matzko, S., Clarke, P.J., Gibbs, T.H., Malloy, B.A., Power, J.F. and Monahan, R. (2002), "Reveal: A Tool to Reverse Engineer class diagrams" in proceedings of 40th International conference on Tools Pacific: Object for Internet, Mobile and Embedded Applications, Sydney, 2002, pp: 13-21.

[11] Hansen, G.W. and Hansen, J.V. (1996), Database Management and Design, 2nd Ed., Prentice-Hall.

Vinita earned her B.A. (H) maths from Delhi University, Delhi, India in 1997. Then she received her Master in computer applications (MCA) degree from IGNOU, Delhi, India in 2000. Now she is perusing her M.Phil. (Computer Science) from Madurai Kamraj University, Madurai, India. She is currently working as lecturer, Dept of Computer Science, Keshav mahavidyalaya, Delhi University, Delhi, India. She has guided 4 B.Sc. (H) CS students for their research work and 10 PGDCA students for their project work. Her area of research and interest are Computer Networks, Astificial Intelligence, Data Mining, Software engineering, Algorithm and DBMS.

The second paragraph uses the pronoun of the person (he or she) and not the author's last name. It lists military and work experience, including summer and fellowship jobs. Job titles are capitalized. The current job must have a location; previous positions may be listed without one. Information concerning previous publications may be included. Try not to list more than three books or published articles. The format for listing publishers of a book within the biography is: title of book (city, state: publisher name, year) similar to a reference. Current and previous research interests ends the paragraph.

Devendra K. Tayal was born in 1977 in Delhi, India. He has acquired the degrees of B.Sc. (H) Maths, M.Sc.(Maths), M.Tech(Computer Engg) & Ph.D from Jawaharlal Nehru University, Delhi, India. He is Assistant Professor in Department of Computer Engg, Jaypee Institute of Information Technology University, Noida, UP, India. He also worked as a lecturer in University of Delhi for about three years. Mr. Tayal has published about a dozen papers in Fuzzy Logic and Software Engg. His areas of interest include Theory of Computation, Algorithms, Software Engg, DBMS, Data Mining and Fuzzy Logic.