Empir Software Eng (2007) 12:575–592
DOI 10.1007/s10664-007-9043-7

An empirical study of a reverse engineering method for the aggregation relationship based on operation propagation

Dowming Yeh · Pei-chen Sun · William Chu · Chien-Lung Lin · Hongji Yang

Published online: 30 May 2007

Editor: Mark Harman

Abstract One of the major obstacles in reverse engineering legacy object-oriented systems is the identification of aggregation relationships. An aggregation relationship, also called whole–part relationship, is a form of association relationship where an object is considered as a part of another object. This characteristic is mostly of semantic nature; therefore, it is difficult to distinguish aggregation from association relationships by implementation mechanism. Most reverse engineering methods for aggregation relationships are based on the lifetime dependence of an object on another object since many implementations of aggregation relationships result in such dependence. However, research literature shows that lifetime dependence is not really a primary property of the aggregation relationships. A reverse engineering approach is proposed on the basis of a primary characteristic for aggregation relationship—propagation of operations. To compare the propagation-based method with the lifetime-based method, we apply both methods to ten class libraries, collect their output, and perform statistical analysis to determine the effectiveness of the two methods. The analysis results show that the propagation-based method performs significantly better than the lifetime-based method and by combining both methods simultaneously the complete aggregation relationships can be uncovered for the class libraries in our experiment.

Keywords Reverse engineering · Aggregation relationship · Object-oriented · Empirical method · Software reengineering

D. Yeh (✉) · P.-c. Sun
National Kaohsiung Normal University, Kaohsiung, Taiwan, Republic of China
e-mail: dmyeh@nknucc.nknu.edu.tw

W. Chu
TungHai University, Taichung, Taiwan, Republic of China

C.-L. Lin
Siliconware Precision Industries, Taichung, Taiwan, Republic of China

H. Yang
De Montfort University, Leicester, UK

1 Introduction

Object-oriented technology is widely adopted in the industry because of its potential advantages to reduce time and cost of software development through the reuse of artifacts produced in various stages of the software lifecycle (Port and McArthur 1999). Consequently, more and more object-oriented systems are deployed into operations and become legacy systems that need to be maintained. Some features of object-oriented technology pose special challenges for the maintenance of object-oriented applications. The functionality in OO systems comes from the cooperation of objects and their methods. Discovering relationships and interactions of these objects is not easy (Richner and Ducasse 1999). There is evidence which suggests that programmers have difficulty in maintaining object-oriented software because of inheritance depth and conceptual entropy of class hierarchies (Daly et al. 1996; Sheldon et al. 2002).

Although certain good software engineering techniques such as encapsulation are employed in these object-oriented systems, other software engineering practices important to the maintenance work, documentation in particular, is often overlooked. Although the modeling consistency in the object-oriented development lifecycle suggest that maintaining object-oriented system documentation is seemingly easier than maintaining procedure-based legacy system documentation. However, an experimental study indicates no strong evidence regarding higher maintainability of object-oriented design documents over structured design documents (Briand et al. 2001). When the original designer is no longer responsible for maintaining his system, whoever assigned to support the system is very likely faced with the situation that source code is the only reliable documentation. Furthermore, the result also suggests that object-oriented design documents are more sensitive to poor design practices. Thus reverse engineering technology is essential for these object-oriented systems. Reverse engineering is the process of analyzing a system to identify the system's components and their interrelationships and create system documentation in different levels of abstraction (Chikofsky and Cross 1990; Baxter and Mehlich 2000).

A fundamental design document in object-oriented system is a class diagram. Class diagrams describe the data and methods of some given classes and the relationships among these classes. The basic relationships include inheritance, association and aggregation. To identify classes and inheritance relationships in the source code is trivial since the syntax of all object-oriented languages marks these concepts distinctly. However, it is arduous to distinguish aggregation relationships from association relationships even for the state-of-the-art technology. When one tries to produce class diagrams from source code by applying CASE tools with reverse engineering facilities, the association relationships in the diagrams may be recognized as aggregation relationships or the other way around. It is semantically correct to recover aggregation relationships as association relationships since an aggregation must be an association since an aggregation relationship is a special kind of an association relationship (Booch et al. 1999). But the aggregation relationship is very important in many domains and should be manifested whenever possible (Barbier et al. 2003). Distinguishing aggregations from associations in an object-oriented model can guide programmers in their implementation work as well as the understanding of the system (Motschnig-Pitrik and Kaasboll 1999).

Moreover, class diagrams of an object-oriented system of a fair size may be hard to comprehend by programmers. Some researchers propose to use design patterns as a logical basis for reverse engineering of object-oriented software systems by recognizing occurrences of known design patterns in source code (Balanyi and Ferenc 2003; Niere et al.

2002; Tonella and Antoniol 2001; Prechelt and Kramer 1998). Design patterns are originally presented by Gamma et al. to represent a high level of abstraction that repeatedly adopted in good quality software design (Gamma et al. 1995). Aggregation relationships are also very frequently employed in design patterns. Therefore, to recover design patterns in existing systems, the reverse engineering of aggregation relationships is an essential step.

The difficulty in distinguishing aggregation from association arises from the nature of the aggregation relationships. An association relationship is the most common relationship between two classes. It denotes that an instance of a class *uses* or *knows* an instance of another class. An aggregation relationship, also called part–whole relationship, further denotes that one of the two instances in an association relationship is *a part of* the other instance. That is, a part is one of the segments or portions into which something is regarded as divided; a part is less than a whole; together, parts constitute a whole. This characteristic of an aggregation relationship is mostly of semantic nature; therefore, it is difficult to distinguish aggregation from association relationships by implementation mechanism.

To distinguish an aggregation relationship from an ordinary association relationship, most reverse engineering methods are based on the analysis of lifetime dependence of an object on another object since many implementations of aggregation relationships result in such dependence. However, the linkage of these two concepts is questionable. Take the composite pattern from the book "design patterns" by Gamma et al. as an example (shown in Fig. 1) (Gamma et al. 1995). From the description of composite pattern, there is an *add* operation that could add new components to the composite object. This implies that a part (component class) could be created independently from the whole (composite class) and then be added to the latter. Although the destruction of a part may be dependent on the whole, such dependence may not show explicitly in some systems such as Java. Therefore, the aggregation relationship in composite pattern can not be identified by lifetime-based approaches. In order to devise a better scheme for uncovering aggregation relationships, true characteristics of aggregation relationships need to be studied thoroughly. With the growing importance of its role in object-oriented modeling, the primary characteristics of aggregation relationships are thoroughly discussed only recently (Barbier et al. 2003).

In this paper, a different reverse engineering approach is proposed on the basis of a primary characteristic of aggregation relationships—*propagation of operations*. Characteristics of aggregation relationships are presented in Section 2. Section 3 summarizes the lifetime-based approaches and shows their pitfalls. The proposed method is described in Section 4. Statistical comparisons of the new method with the lifetime-based method are discussed in Section 5. Some conclusive remarks are made in the final section.

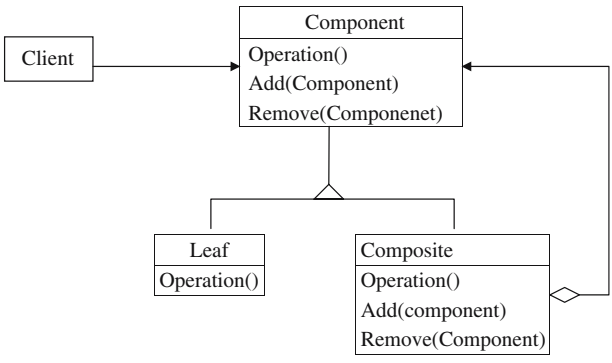


Fig. 1 Composite pattern (in OMT notation)

2 Properties of Aggregation Relationships

Saksena et al. proposed three primary characteristics for aggregation relationships (Saksena et al. 1998). The first characteristic concerns the static structure of the aggregation relationship including transitivity, antisymmerty, and irreflexivity. Transitivity means that if A is a part of B, and B a part of C, then A is a part of C. Both antisymmerty and irreflexivity properties address the instance level. Asymmetry property states that the whole cannot be a part of the part, while irreflexivity asserts that an object cannot be both the whole and the part at the same time. The second characteristic concerns the lifetime binding relationships between the whole object and the part object. The ownership property of the whole on the part is the third characteristic. This earlier belief that lifetime dependence is a primary concept reflects the reason why it becomes the major route for reverse engineering aggregation relationships. However, lifetime dependence is found to be secondary characteristics by later studies.

Henderson-Sellers and Barbier pointed out that lifetime dependence is not a primary property of the aggregation relationship along with ten other secondary properties including separability (Henderson-Sellers and Barbier 1999). These secondary properties represent different partitions of aggregation relationships. For the case of lifetime dependence, the part can be created before, at the same, or after the birth of the whole, and the part can also be destroyed before, at the same, or after the death of the whole. Therefore, there are nine possible cases to partition the set of aggregation relationships (Barbier et al. 2003).

Emphasizing semantic constraints and perspectives from the cognitive science, Motschnig-Pitrik and Kaasbol also categorize the aggregation relationships in the degree of dependence (Motschnig-Pitrik and Kaasboll 1999). The following three dependence sub-constraints are defined:

1. Creation-implication: the part must be created after or concurrently with the whole.
2. Immutability: the part may not be separated from the whole during its lifetime.
3. Deletion-implication: the part must be deleted before or concurrently with the whole.

A lifetime dependent aggregation relationship must satisfy all three sub-constraints. But some aggregation relationships may not satisfy some of the sub-constraints. For example, the aggregation relationship between a heart and its associated body only satisfies the creation-implication sub-constraint. All possible combination of the three sub-constrains partitions aggregation relationships into seven categories as shown in Table 1. The notation ○ denotes that the sub-constraint applies in the particular type of aggregation relationship.

Table 1 Categories of aggregations

Sub-constraint	Creation-implication	Immutability	Deletion-implication
Category			
Lifetime dependent	○	○	○
Essential and immutable	○	○	
Essential	○		
Non-separable		○	○
Optional and immutable		○	
Deletion dependent			○
Optional			

The category satisfying creation-implication and deletion-implication is found irrelevant (Motschnig-Pitrik and Kaasboll 1999).

An aggregation relationship satisfying the creation-implication sub-constraint is called essential, while an immutable aggregation relationship satisfy the immutability sub-constraint. An essential and immutable aggregation relationship is exemplified by the situation of Leaf part-of Tree. A non-separable aggregation results when a part object is created independent of a whole object, but a part object may not be disconnected from a whole object once it is inter-connected with the whole. An example is that a chip is a non-separable part of a motherboard. An optional or independent relationship may exist when a part objects is not linked to any whole object. For example, a wheel and a car comprise an optional aggregation relationship. An optional and immutable relationship can be exemplified by a person and a couple. Finally, a node may be created independently of a linked list and interconnected with different linked lists during its lifetime, but the deletion of the linked list may cause the deletion of the node; therefore, a deletion dependent relationship results.

Since lifetime dependence cannot be a reliable source for extracting aggregation relationships, the primary properties of aggregation must be investigated so that a better reverse engineering scheme can replace or complement the lifetime-based approach. The previous two studies presented more primary properties as listed below:

- Resultant: a property that can be deduced from explorations of attributes of the parts. A consequence of this property is message propagation from the whole to the part in order to accumulate the related information from the part (Henderson-Sellers and Barbier 1999).
- Emergent: a property that cannot be deduced from properties of the parts (Henderson-Sellers and Barbier 1999).
- Directedness: the part-of relationship is directed from the part to the whole; while the reverse relationship, has-part, is directed from the whole to the part (Motschnig-Pitrik and Kaasboll 1999).
- Tendency to form (de-)composition hierarchies: this is computationally relevant for propagation of operations, including the propagation of deletion operation. For example, to draw a compound GUI object would propagate *draw* operation to its constituting parts (Motschnig-Pitrik and Kaasboll 1999).

Both the resultant property and the tendency to form hierarchies indicate that the message or operation propagation is manifest in an aggregation relationship. Moreover, such propagation could be described syntactically. Therefore, it is possible to design a reverse engineering method based on the propagation of operations between the part and the whole.

3 Lifetime-based Reverse Engineering Methods

As discussed in the previous section, any particular form of lifetime dependence between two classes covers only some categories or partitions of the complete set of aggregation relationships. Consequently, a lifetime-based reverse engineering method can only recover aggregation relationships satisfying its specific lifetime dependence criteria. A design pattern recovery system by Prechelt and Kramer deduce aggregations from a data member declaration of non-reference type in a C++ class based on the fact that the data element is created and destroyed with the containing class (Prechelt and Kramer 1998). Antoniol and

his colleagues maintain a similar view with the extension to include references (or pointers) to template or object array data (Antoniol et al. 2001) Such extension obviously cover more aggregation relationships. From the given examples, they seem to recognize a creation operation such as *new* to follow the declarations of templates and arrays to allocate their space. These kind of lifetime-coincident methods merely recover one category of the aggregation relationship, namely, the lifetime dependent category in Table 1.

Other kinds of lifetime-based methods are possible. Seeman and Gudenberg propose that an association be an aggregation if the referenced object (the part) in a class (the whole) is also created by the class (Seemann and Gudenberg 1998). How the part is created is not defined, however. A non-reference declaration as well as an explicit creation operation such as *new* seems acceptable. The dependence of the deletion of the part on the destruction of the whole is not defined since their work is focused on Java systems whose garbage collection facility destroys objects implicitly. The proposed method is clearly based on the creation-implication criteria, and covers three categories of aggregation relationship, lifetime dependent, essential, and essential and immutable. The other four categories are still not considered as candidate aggregation relationships.

In the Columbus tool, aggregation is further refined into composite aggregation (or composition) and regular aggregation (Balanyi and Ferenc 2003). The concept of composition in source code is defined as a data member of the part class in the whole class, thus corresponding to the lifetime dependent category. This is consistent with the definition of composition given by OMG that requires that the whole is responsible for the creation and destruction of its parts (Object Management Group 2001). The concept of regular aggregation relationship in Columbus is defined as a pointer or a reference to the part class in the whole class without other criteria. This definition would prevent the restriction of lifetime analysis, but on the other hand many association relationships would become candidate aggregation relationships, thus reducing the accuracy of the reverse engineering result.

Three reverse engineering criteria are proposed by Di Lucca et al. to identify candidate aggregation relationships in a data-intensive legacy system (Di Lucca et al. 2000):

1. The lifetime of two classes of objects coincides, or
2. The creation or the deletion of two classes coincides, and each reference to a class results in a reference to the other class, or
3. The creation or the deletion of two classes coincides, or each reference to a class results in a reference to the other class (if the candidate part object is exclusively owned by the whole object, the relationship is further refined to compositional aggregation.)

The candidate relationships from the last criterion seem to cover those from the first two criteria. The method requiring either creation-implication or deletion-implication is still limited to five categories and the other two categories, namely, optional and immutable, and optional, cannot be covered. In fact, the lifetime-based methods would not likely resolve the two optional categories of aggregation relationships since the immutability is not related to lifetime in any way.

The reverse engineering method for aggregation is not clearly defined in some work. Niere et al. discusses the reverse engineering of composite pattern where an aggregation relationship is a major characteristic (Niere et al. 2002). However, there is no explicit rule requiring an aggregation relationship in the definition of composite pattern reverse engineering rule. Instead, an association relationship and a delegation pattern occur between two operations of the classes with the association relationship.

4 A Propagation-based Reverse Engineering Method

Appropriate reverse engineering criteria should be derived from the primary properties of aggregation relationships. However, most of the primary properties of aggregation relationships discussed earlier are semantic. The only property found manifest in the implementation mechanism is the propagation of operations or message propagation. Such property implies a resultant property and a decomposition hierarchy. Hence, we propose the following reverse engineering method based on propagation of operations.

4.1 Reverse Engineering Criteria

Let ClassP and ClassW be two distinct classes in the source code under analysis, ClassP and ClassW may have aggregation relationship with ClassP as the part and ClassW as the whole if

1. There is a data member declaration involving ClassP in ClassW, and
2. There exists a call to an operation of ClassP in the definition of some operations of ClassW, indicating a possible propagation of operation.

A data element declaration involving ClassP in ClassW could be a pointer type to classP, an array of ClassP or even a class template involving classP. Hence the creation of ClassP (the part) could be independent from that of ClassW (the whole). The first condition is essentially the criteria adopted by the Columbus tool (Balanyi and Ferenc 2003). This condition requires that the relationship between ClassP and ClassW is more than a temporary one. It excludes cases like a declaration of ClassP in a method for instances of ClassW and declaration of ClassP as a formal parameter of a method for ClassW. Both indicate a local reference to ClassP during the execution of a method in ClassW. Though the lifetime of the part and the whole may not have any dependence relations as discussed earlier, their aggregation relationship should persist considerably longer than the invocation of a method. Otherwise, such a short-life relationship is very unlikely to be an aggregation relationship worthy of reverse engineering effort.

The operation propagated in the second condition includes overloaded operators. Although overloaded operators look syntactically different from a usual operation call, it is still an operation call semantically. The propagated operations considered in the second condition exclude creation operations such as *new* and deletion operations such as *delete*. Both operations can be legitimately considered as propagated operations, but the exclusion would make a clear distinction of our method from the lifetime-based method. The effectiveness of our method would not suffer from such exclusion since the resultant property of aggregation relationship implies that the propagated operation should include other messages besides creation and deletion. Therefore, the classes that satisfy condition 1 but not condition 2 are those that access directly attributes of ClassP objects without passing ANY messages to ClassP objects (except maybe *new* or *delete*).

We may examine the appropriateness of our method by applying the criteria to source code implementing the composite pattern in Fig. 1. The source code is adopted from the book by Gamma et al. and shown in Fig. 2 after some abridgement and simplification (Gamma et al. 1995). There is a declaration of an array of pointers to objects of class *Equipment*, _equipment, as one of the data elements in class *CompositeEquipment*. And in one of the definition of the operations of *CompositeEquipment*, *NetPrice*, there is a call to the *Netprice* operation of *Equipment*. Thus from our definition, an aggregation relationship

Fig. 2 An example of operation propagation

```
class Equipment {
  public:
      virtual Currency NetPrice();
      virtual void Add(Equipment*);
      virtual void Remove(Equipment*);
  . . .
  };
  class CompositeEquipment : public Equipment {
   public:
      virtual Currency NetPrice();
      virtual void Add(Equipment*);
      virtual void Remove(Equipment*);
  . . .
  private:
      Equipment* _equipment[MAX_EQUIP];
  };
  Currency CompositeEquipment::NetPrice () {
      Currency total = 0;
        int i = 0
      for (i = 0; i < component_number; i++) {
          total += _equipment[i]->NetPrice();
      }
      return total;
}
```

may exist between the class *CompositeEquipment* and the class *Equipment*. In this example, the names of the propagated methods are the same for the whole and the part (usually called *delegation*), the chance of the relation being an aggregation is even stronger. This is a clear indication of the resultant property. However, we do not require propagated operations to be delegation in our current approach.

4.2 Implementation

To test the propagation-based method, we implemented a prototype tool in Visual Basic. The input to the tool is a set of C++ source code files. The output is a list of candidate aggregations with the whole class and the part class identified. The analysis is performed in a straightforward way without involving unnecessary type inference. The first step is to collect the declarations of all the classes, their attributes, and their methods. Secondly, bodies of the methods in each class are examined to uncover any call to methods of some other classes. If such a call is found and the object whose method is called is declared in a data member, a candidate aggregation relationship is marked with class name of the calling object as the whole and class name of the called object as the part.

Class name of the called object is usually the same as the one declared in the calling class. However, the declaration in the calling class may not be the actual class name due to polymorphism and language features in C++ such as *typedef* and class template. For example, a weakly typed data structure called *List* is adopted the original source code for the composite pattern instead of the pointer array. The *List* is basically a linked list whose nodes contain pointers to the class *Object*, which is the class inherited by all other classes. Therefore, our tool looks up the list collected in the first step to check whether the called method is truly in the set of methods of the declared class. If it is not, the actual name of the called class is searched for in the list of classes with method name matching the called method. The result of the search may contain more than one class since it is possible that more than one class provide methods with the same name. Should this happens, the actual

class name needs to be resolved manually. From our experience, such manual identification rarely happens. An automated solution to infer proper types would require complicated type inference algorithms (Tonella and Potrich 2001; Duggan 2001). Our method can be easily implemented in any rule-based reverse engineering and program understanding systems.

Any aggregation relationships involving classes in the C++ standard class libraries are neglected because their source code cannot be accessed directly. Moreover, such aggregation relationships are much less important than aggregation relationships between user-defined classes. There are some cases when a user-define class is contained in a standard class. These "embedded" user-defined classes are still covered by our tool. So, for example, for a data member declaration Vector<application> app in a class A, the aggregation relationship between class A and Vector template class is not considered, but the aggregation relationships between class A and class application is still included in the result.

4.3 A Case Study

The set of candidates resulted from the propagation-based analysis should include that from the lifetime-based method theoretically since the propagation of operation is indeed a primary property of aggregation relationships. However, such inclusion does not occur in practice because of certain features of object-oriented languages. This fact is illustrated through a case study where both methods are applied to the same class library (Yeh and Kuo 2002). The name of the library is Oz++, and the result is shown in Table 2.

Most of the candidates identified exclusively by the propagation-based method (noted by the first, second, fifth, and sixth rows in Table 2) are complex data structures such as binary trees and GUI structures. This is consistent with the analysis of Motschnig-Pitrik and Kaasboll, where a major category is associated with an example of a node being part of a data structure (Motschnig-Pitrik and Kaasboll 1999). This category exhibits only deletion-implication characteristic, thus the lifetime-based approach often overlooks this kind of aggregation relationships. Because Oz++ is a GUI class library, the number of such aggregation relationships may be comparatively more than that in other types of object-oriented systems. Nevertheless, it illustrates a major strength of the propagation-based method.

On the other hand, the parts are usually "simple" objects in the candidates resolved exclusively by the lifetime-based method, for example, SimpleString objects. The propagation-based method cannot detect these relationships because the whole accesses attributes of the part directly without sending a message possibly to avoid the overhead of message passing. With a pure object-oriented implementation such as SmallTalk, where all objects must interact through messages, these aggregation relationships would be covered

Table 2 Some analysis result of Oz++ class library

Whole	Parts	Lifetime	Propagation
BinaryTree	BinaryNode		○
Icon	CompoundString		○
Icon	SimpleString	○	
InetAddress	SimpleString	○	
Queue	ListEntry		○
ScrolledListBox	ListBox		○

by the propagation-based method as well. However, in the case of C++ and Java where direct manipulation of attributes from other classes is allowed, to extract more aggregation relationships propagation-based method may be supplemented with lifetime-based approach.

5 Assessment of the Propagation-based Method

We applied both the propagation-based method and the lifetime-based method to ten C++ class libraries and compare the effectiveness of the two methods. These ten class libraries are selected from the search results of Google search engine with keywords "C++ class library." The first ten class libraries with downloadable source code were chosen as the sample of this empirical study. The names of the libraries are Botan, Cgicc, common C++, Cppima, DosTMUit++, glObs, OOMPI, MV++, Oz++, and SCATMECH.

Botan implements a variety of cryptographic algorithms and formats. Cgicc is an ANSI C++ compliant class library that greatly simplifies the creation of CGI applications for the World Wide Web. Common C++ is a framework offering portable support for threading, sockets, file access, and other system services. Cppima is designed for image processing. DosTMUit++ is a DOS Text mode user interface toolkit. GlObs is a software package that allows people to create 3D applications and games. OOMPI is a class library specification that encapsulates the functionality of Message Passing Interface. MV++ is a set of concrete vector and simple matrix classes for numerical computing. Oz++ is a class library for Motif (1.2 or later) on X Window System. SCATMECH is developed to distribute bi-directional reflectance distribution function (BRDF) models for light scattering from surfaces. The numbers of classes and lines of code (LOC) in the libraries are shown in Table 3. The counting of LOC includes comments.

5.1 Effectiveness Measures

The effectiveness of a reverse engineering method should be analyzed in two dimensions of information retrieval, recall and precision. They are also named as coverage and accuracy in Rugaber et al. 2001. Recall concerns the extensiveness of the search for candidate items so that more design concepts may be identified; while precision addresses the ratio of the actual items in the candidate set so that the reverse engineering effort may be reduced. We adopt the metrics, *adequacy*, proposed by Di Lucca et al. to express precision of the

Table 3 Numbers of classes and lines of code

Class library	Number of classes	LOC(.cpp files)	LOC(.h files)
Botan	228	37,392	8,349
Cgicc	22	4,122	4,849
Common C++	172	30,131	17,950
Cppima	43	3,992	2,734
dosMUit++	135	14,645	5,321
Globs	11	2,259	543
Mv	14	6,846	3,249
Oompi	26	11,916	5,059
Oz++	144	7,738	10,246
scatmech	112	13,477	5,381

methods (Di Lucca et al. 2000). Precision shows the effectiveness to recover relationships that are meaningful conceptually in the application domain. It is defined as follows:

$$\text{Precision} = |N|/|M| \times 100\%$$

Where M is the set of candidate aggregation relationships recovered by the method, and N is the subset of M that can be associated with a concept in the application domain, that is, actual aggregation relationships.

The recall is the number of concepts uncovered by a reverse engineering method versus the total number of concepts. With U denoting the complete set of aggregation relationships of a system, the recall is defined as follows:

$$\text{Recall} = |N|/|U| \times 100\%$$

The complete set of aggregation relationships of a system can be derived from its accompanying class diagrams. Unfortunately, such diagram is often missing or incomplete. This is certainly the case for the ten libraries under study. Theoretically, the complete set of aggregation relationships in a class library can be uncovered by the propagation-based method since propagation of operation is a primary property of aggregation, but propagation of operation may be replaced with direct access of attributes as discussed earlier. The case of direct access of data member with non-reference type is covered by the lifetime-based approach. Therefore, the only kind of aggregation relationships that is not included in the union of the sets derived by the propagation-based method and the lifetime-based method is that implemented with the part declared with a pointer or a reference and the whole directly accessing attributes of the part without propagating operations. After a careful examination of the ten class libraries, we found that no aggregation is implemented in this way, so the complete set of aggregation relationships equals the union set of both methods.

The judgment of whether candidate aggregation relationships is true aggregation or not is based on the domain knowledge and documentation of these class libraries. Domain knowledge involves the semantic content of a class library. Some libraries such as Oz++ and MV++ involve general knowledge in GUI and mathematics, while others require more expertise to distinguish actual aggregations from the candidate ones. In the case of the Botan library, we resort to a local computer security expert for various relationships among cryptography objects. The other source for recovering actual aggregations is to study the documentation associated with these class libraries. Documentation includes design documents, comments in the code, readme files, and so on. Aggregation relationships may be inferred from keywords such as contain, comprise that connote that a class is part of another class. However, such inference needs to be conducted with care since topological inclusion is not aggregation (Barbier and Henderson-Sellers 2001). For example, in the documentation of the Cgicc class library, the comment "*Get the HTMLElementList embedded in this elemen*" is found for the class *HTMLElement*. This implies an aggregation relationship between *HTMLElement* and *HTMLElementList*. On the other hand, a sentence like "*the attributes (HTMLAttributeList) associated with this element*" denotes a regular association relationship.

5.2 Comparison of Different Minimum Numbers of Operation Propagated

In our initial definition, there is no limitation on the number of operations propagated from the candidate whole class to its part class. This definition may result in a certain amount of

association relationships being considered candidate aggregation relationships, even though the first condition of the definition filters some of these association relationships. Therefore, the effect of the minimum number of the operation propagated is studied to explore the possibility of raising this number as a filter to eliminate association relationships from becoming candidates. Note that we merely require that the number of propagated operations occurs more than once without considering what operation it is; therefore, it includes the case that the same operation of the part class is called twice in the source code. More formally, the second condition of the propagation-based method is modified as follows:

> There exist at least two occurrences of calls to some operations of classP in the definition of some operations of ClassW, indicating a possible propagation of operation.

With the minimum number of operation propagated increased by one, the precision of the propagation-based method is expected to improve while the recall may be affected to a minimum degree hopefully.

Let α_1 and ν_1 be the average of precision values and recall values for all libraries of the propagation-based method requiring only one propagated operation, and α_2 and ν_2 be the average of precision values and recall values for all libraries of the propagation-based method requiring at least two operations propagated. The following two research hypotheses are formed:

Hypothesis 1 The propagation-based method requiring minimum two propagated operators produces significantly higher precision value than the propagation-based method requiring only one. That is, $\alpha_2 > \alpha_1$.

Hypothesis 2 There is no significant difference between ν_1 and ν_2.

To compare the effectiveness of propagation-based methods with two different numbers of propagated operations, the appropriate statistical analysis model is the Student's *t* distribution (Juristo and Moreno 2001). The t distribution is referred to analyze the difference between means for small samples. The following two null statistical hypotheses are formed with respect to our research hypotheses:

H^0 1 There is no significant difference between α_1 and α_2.
H^0 2 There is no significant difference between ν_1 and ν_2.

The results of applying the two variants on the class libraries are shown in Table 4 and the statistical result of the paired-samples t test using SPSS is shown in Table 5.

Both null hypotheses are rejected. This indicates that the precision value of the method requiring a minimum number of two propagated operations, 74.45%, is significantly better than the one without limitation on the number, 70.65%; While the recall value of the latter, 90.6%, excels the former by over 9%. Therefore, by requiring a minimum number of propagated operations, some association relationships are filtered so that the precision of the propagation-based method is improved significantly but with the side effect of also sacrificing a significant number of aggregation relationships. We consider this an overall improvement since the filtered aggregation relationships are characterized by a single method being propagated along such relation. The importance of these relationships may be weaker at least from the viewpoint of software maintenance since the coupling of the two classes is low and efforts needed during testing and maintenance are consequently less significant (Chidamber and Kemerer 1994). On the other hand, the improvement in the precision will definitely reduce the effort of reverse engineering. If the recall is of primary

Table 4 Analysis results of different numbers of operation propagated

Class library	Total number of aggregation	Number of propagate ≥ 1				Number of propagate ≥ 2			
		$\|M\|$	$\|N\|$	Recall (%)	Precision (%)	$\|M\|$	$\|N\|$	Recall (%)	Precision (%)
Botan	49	79	38	77.55	48.10	63	34	69.39	53.97
Cgicc	2	2	2	100	100	2	2	100	100
Common C++	18	21	13	72.22	61.9	20	13	72.22	65
Cppima	7	18	7	100	38.89	14	6	85.71	42.86
dosMUit++	27	54	26	96.3	48.15	39	24	88.89	61.54
Globs	9	11	7	77.78	63.64	8	6	66.67	75
Mv	7	7	7	100	100	7	7	100	100
Oompi	13	14	13	100	92.86	13	12	92.31	92.31
Oz++	19	34	18	94.74	52.94	26	14	73.68	53.85
scatmech	32	28	28	87.5	100	21	21	65.63	100
Average value	18.3		15.9	90.6	70.65		13.9	81.45	74.45

concern, one may consider relaxing the limitation on the minimum number of operation propagated, but the precision of the propagation-based method would suffer significantly.

5.3 Comparison with Lifetime-based Method

Here the lifetime-based method means creation-implication as suggested by Seemann and Gudenberg; therefore, the part object is either a non-reference type or a reference/pointer type with its corresponding *new* operation appearing in *every* constructor of the candidate whole class. However, there is no occurrence of a part with reference/pointer type found in our analysis result. Therefore, all candidate aggregation relationships extracted by the lifetime-based method in the following analysis fall into the lifetime dependent category. The propagation-based method is represented by the variant with a minimum number of two propagated operations. Let α_p and ν_p be the average of precision values and recall values for all libraries of the propagation-based method, and α_l and ν_l be the average of precision values and recall values for all libraries of the lifetime-based method, respectively. The propagation-based method is expected to recover more aggregation relationships than the lifetime-based approach, but maintain a comparable level of precision value with the lifetime-based method. Thus the following hypothesis is formed:

Hypothesis 3 The propagation-based method produces higher recall value than the lifetime-based method. That is, $\nu_p > \nu_l$.

Hypothesis 4 There is no significant difference between α_p and α_l.

Table 5 Paired-samples *t* test of two propagation-based methods

Hypothesis test	Mean(2 or more propagated operations)	Mean(1 propagated operation)	*t* Value	*df*	*P* value(two-tailed)
$H^0$1(precision)	0.7445	0.7065	2.41	9	0.039*
$H^0$2(recall)	0.8145	0.9061	−3.57	9	0.006*

*$p<0.05$

To compare the propagation-based method with the lifetime-based method, the t-test is used again. Again two null statistical hypotheses are formed:

H^0 3 There is no significant difference between ν_p and ν_l.
H^0 4 There is no significant difference between α_p and α_l.

The results of applying both methods on the class libraries are shown in Table 6. Note that the precision value of the lifetime-based method for the Cppima library is not defined since the denominator in the formula is zero. Therefore, the values from Cppima library are excluded from the subsequent statistical analysis, resulting in the degree of freedom for the t distribution decreased by one. The statistical result of the paired-samples t-test using SPSS is shown in Table 7.

The null hypothesis H^0 3 is rejected, while hypothesis H^0 4 cannot be rejected. The recall of the propagation-based method, averaging over 80%, is significantly better that that of the lifetime-based method, about 46%. The average precision values of both methods are about the same, 77.96 versus 65.58%. Therefore, the propagation-based method is significantly more effective than the lifetime-based method since it recovers significantly more aggregation relationships while maintains the same level of precision with the lifetime-based method if the minimum number of operation propagated is restricted to be at least two. If the limitation on the minimum number of operation propagated is relaxed to be one, the propagation-based method still excels significantly in recall, but it is less adequate than the lifetime-based method. The result also indicates that the percentage of aggregation relationships that fall into the lifetime dependent category is a little over 50%, implying that only about half of the total aggregation relationships are uncovered by the traditional lifetime-based method.

5.4 Validity of the Experiment

The validity of a controlled experiment can be categorized into internal validity and external validity. Internal validity is the validity of inferences about whether observed co-variation between the presumed treatment and the presumed outcome reflects a causal relationship; while external validity is about the degree to which a causal relationship holds

Table 6 Analysis results of two methods

Class library	Total number of aggregation	Propagate-based				Lifetime-based			
		$\|M\|$	$\|N\|$	Recall (%)	Precision (%)	$\|M\|$	$\|N\|$	Recall (%)	Precision (%)
Botan	49	63	34	69.39	53.97	24	17	34.69	70.83
Cgicc	2	2	2	100	100	1	0	0	0
Common C++	18	20	13	72.22	65	21	15	83.33	71.43
Cppima	7	14	6	85.71	42.86	0	0	0	–
dosMUit++	27	39	24	88.89	61.54	16	13	48.15	81.25
Globs	9	8	6	66.67	75	10	6	66.67	60
Mv	7	7	7	100	100	7	7	100	100
Oompi	13	13	12	92.31	92.31	6	6	46.15	100
Oz++	19	26	14	73.68	53.85	6	4	21.05	66.67
scatmech	32	21	21	65.63	100	50	20	62.5	40
Average value	18.3		13.9	81.45	77.96		9	46.25	65.58

Table 7 Paired-samples *t* test of propagation-based and lifetime-based method

Hypothesis Test	Mean(propagate-based)	Mean(lifetime-based)	*t* Value	*df*	*P* value(two-tailed)
$H^0$3(recall)	0.8145	0.4625	2.945	9	0.016*
$H^0$4(precision)	0.7796	0.6558	0.909	8	0.390

*$p<0.05$

over variations in subject, task, and environment (Sjoberg et al. 2005). The major threat to our experiment is the external validity.

The selection of class libraries may affect the analysis result. A common way to reduce such threat is to randomly select class libraries for the experiment from a large pool of class libraries with no particular characteristics. The way we select class libraries is very similar to such a process. The class libraries accessible on the Web should cover almost all kinds of libraries. Though the ten libraries are not randomly selected, instead they are picked based on the order of search result from Google in January, 2002. The deliberate selection is still avoided, and the threat to validity reduced.

Another threat to external validity is that there is no occurrence of a part with reference/pointer type found in the class libraries in this experiment. It is possible that the experimental results are specific to systems that satisfy this condition.

There is also an internal validity threat in the classification of candidate aggregation relationships into true aggregation and simple association. Besides taking advantage of domain knowledge and related documentation as described in Section 5.1, the classification is conducted by two of the authors independently to reduce possible errors. Contradictory opinions are resolved through discussion of the two classifiers.

6 Conclusions

The reverse engineering of aggregation relationships is better performed based on the propagation of operations than on the traditional lifetime dependence. The propagation of operation is a major characteristic of the aggregation relationship, while the lifetime dependence is not. Aggregation relationships can be categorized in the degree of lifetime dependence, and the lifetime-based approach covers only a subset of all the categories. We define a reverse engineering method based on propagation of operations among objects. To compare this propagation-based method with the lifetime-based method, we apply both methods to ten class libraries, collect their outputs, and perform statistical analysis on the effectiveness of two methods. The experimental results show that the propagation-based method recovers significantly more aggregation relationships than the lifetime-based method while maintaining the same level of precision as the lifetime-based method.

However, direct accesses of attributes may be used instead of propagation of operations in practice causing propagation-based method to fail in extracting all possible aggregation relationships. Such direct access usually occurs when the part is a “simple” part, and the lifetime-based method may extract the associated aggregation relationship provided that the lifetime of the part is dependent on that of the whole. Therefore, lifetime-based method supplements propagation-based method in uncovering aggregation relationships from systems implemented with hybrid object-oriented languages such as C++. Though there is a possible kind of implementation of aggregation relationships not covered by both methods, the set of aggregation relationships can be uncovered completely by applying both methods simultaneously for the class libraries in our experiment.

Acknowledgement This work was partially supported by the National Science Council, Taiwan, R.O.C., under Grant NSC 89-2213-E-020-004- and Department of Industrial Technology, Ministry of Economic Affairs, R.O.C. under the grant 96-EC-17-A-02-S1-029.

References

Antoniol G, Casazza G, Di Penta M, Fiutem R (2001) Object-oriented design patterns recovery. J Syst Softw 59(2):181–196

Balanyi Z, Ferenc R (2003) Mining design patterns from C++ source code. In: Proceedings of the international conference on software maintenance. IEEE Computer Society Press, Silver Spring, MD, pp 305–314

Barbier F, Henderson-Sellers B (2001) The whole–part relationship in object modelling: a definition in cOlOr. Inf Softw Technol 43:19–39

Barbier F, Henderson-Sellers B, Le Parc-Lacayrelle A, Bruel J (2003) Formalization of the whole–part relationship in the unified modeling language. IEEE Trans Softw Eng 29:459–470

Baxter ID, Mehlich M (2000) Reverse engineering is reverse forward engineering. Sci Comput Program 36:131–147

Booch G, Rumbaugh J, Jacobson I (1999) The unified modeling language user guide. Addison-Wesley, Reading, MA

Briand L, Bunse C, Daly J (2001) A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE Trans Softw Eng 27:513–530

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20:467–493

Chikofsky EJ, Cross JH (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 23:13–17

Daly J, Brooks A, Miller J, Roper M, Wood M (1996) Evaluating inheritance depth on the maintainability of object-oriented software. Empirical Software Engineering, An International Journal 1:109–132

Di Lucca GA, Fasolino AR, De Carlini U (2000) Recovering class diagrams from data-intensive legacy systems. In: Proceedings of international conference on software maintenance. IEEE Computer Society Press, Silver Spring, MD, pp 52–63

Duggan D (2001) Finite subtype inference with explicit polymorphism. Sci Comput Program 39:57–92

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, MA

Henderson-Sellers B, Barbier F (1999) What is this thing called aggregation. In: Proceedings of TOOL EUROPE'99. IEEE Computer Society Press, Silver Spring, MD, pp 236–250

Juristo N, Moreno AM (2001) Basics of software engineering experimentation. Kluwer, Boston

Motschnig-Pitrik R, Kaasboll J (1999) Part–whole relationship categories and their application in object-oriented analysis. IEEE Trans Knowl Data Eng 11:779–797

Niere J, Schafer W, Wadsack JP, Wendehals L, Welsh J (2002) Towards pattern-based design recovery. In: Proceedings of international conference on software engineering. IEEE Computer Society Press, Silver Spring, MD, pp 338–348

Object Management Group (2001) OMG Unified Modeling Language Specification-Version 1.4. Object Management Group

Port D, McArthur M (1999) A study of productivity and efficiency for object-oriented methods and languages. In: Proceedings of the sixth Asia Pacific software engineering conference. IEEE Computer Society Press, Silver Spring, MD, pp 128–135

Prechelt L, Kramer C (1998) Functionality versus practicality: employing existing tools for recovering structural design patterns. Journal of Universal Computer Science 4:866–882

Richner T, Ducasse S (1999) Recovering high-level views of object-oriented applications from static and dynamic information. In: Proceedings of the international conference on software maintenance. IEEE Computer Society Press, Silver Spring, MD, pp 13–22

Rugaber S, Shikano T, Stirewalt REK (2001) Adequate reverse engineering. In: Proceedings of international conference on automated software engineering. IEEE Computer Society Press, Silver Spring, MD, pp 232–241

Saksena M, Franceand RB, Larrondo-Petrie MM (1998) A characterization of aggregation. In: Proceedings of OOIS'98. IEEE Computer Society Press, Silver Spring, MD, pp 11–19

Seemann J, Gudenberg JW (1998) Pattern-based design recovery of Java software. In: Proceedings of FSE'98. ACM, New York, pp 10–16

Sheldon FT, Jerath K, Chung H (2002) Metrics for maintainability of class inheritance hierarchies. Journal of Software Maintenance and Evolution: Research and Practice 14:147–160

Sjoberg DIK, Hannay JE, Hansen O, Kampenes VB, Karahasanovic A, Liborg N-K, Rekdal AC (2005) A survey of controlled experiments in software engineering. IEEE Trans Softw Eng 31(9):733–753, Sept

Tonella P, Antoniol G (2001) Inference of object-oriented design patterns. Journal of Software maintenance and Evolution: Research and Practice 13:309–330

Tonella P, Potrich A (2001) Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society Press, Silver Spring, MD, pp 376–385

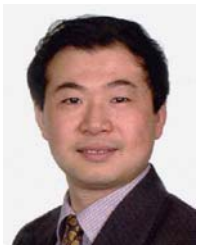
Yeh D, Kuo W (2002) Reverse engineering aggregation relationship based on propagation of operations. In: Proceedings of European conference on software maintenance and reengineering. IEEE Computer Society Press, Silver Spring, MD, pp 223–229

Dowming Yeh is an Associate Professor in the Institute of Information and Computer Education at the National Kaohsiung Normal University, Taiwan, R.O.C. He received his Ph.D. in Computer Science from the University of Utah in USA. Before assuming his current post, he was an Associate Professor in the Department of Management Information System at the National Pingtung University of Science and Technology and a Manager at the Institute for Information Industry in Taiwan. His research interest includes software reengineering, e-Learning, web engineering, program analysis, and human computer interaction. Dr. Yeh is a member of IEEE and ACM.

Pei-Chen Sun is currently an Associate Professor and serves as System Manager at National Kaohsiung Normal University. He holds a Ph.D. in Management Information Systems from National Sun Yat-Sen University. His current research interests include e-Learning, electronic commerce, and knowledge management. His researches have been published in Journal of Information Management, Computers & Education, Journal of Information Science and Engineering, and International Journal of Innovation and Learning, etc.

William Cheng-Chung Chu is a Professor and the Chairman of the Department of Computer Science and Information Engineering at Tunghai University, Taiwan. From 1994 to 1998, he was an Associate Professor at the Department of Information Engineering and Computer Science at Feng Chia University. He was a Research Scientist at the Software Technology Center of the Lockheed Missiles and Space Company, Inc., where he received special contribution awards in both 1992 and 1993. He also received a PIP award in 1993. In 1992, he was also a Visiting Scholar at the Department of Engineering Economic Systems at Stanford University, where he was involved in projects related to intelligent knowledge-based expert systems. His current research interests include software engineering, embedded systems, and E-learning. Dr. Chu received his M.S. and Ph.D. degrees from Northwestern University in Evanston Illinois in 1987 and 1989, respectively, both in computer science. He has edited several books and published over 100 referred papers and book chapters, as well as participating in many international activities, including organizing international conferences.

Chien-Lung Lin received his M.S. degree in Management Information System from the National Pingtung University of Science and Technology. His research interests include web engineering and human computer interaction.



Professor Hongji Yang is Head of Computer Science Division at Schooling of Computing, De Montfort University, England and leads the Software Evolution and Reengineering Group. His general research interests include software engineeering and distributed computing. He served as a Programme Co-Chair at IEEE International Conference on Software Maintenance 1999 (ICSM '99) and is serving as the Programme Chair at IEEE Computer Software and Application Conference 2002 (COMPSAC'02).