Assessing and Restructuring of Classes Based on Cohesion

Heung Seok Chae and Yong Rae Kwon
Department of Computer Science
Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon 305-701, Korea
{hschae,yrkwon}@salmosa.kaist.ac.kr

Abstract

Classes are the fundamental concepts in the object-oriented paradigm. They are the basic units of object-oriented programs, and serve as the units of encapsulation, which promotes the modifiability and the reusability of them. However, improper modeling in the design phase or uncontrolled changes during the maintenance phase can degrade the quality of classes, which leads to programs cumbersome to maintain or extend.

This paper proposes a method for assessing the quality of classes and restructuring them. We define an abstraction as the most cohesive form of a class, and the cohesion indicates the extent to which a class approaches the abstraction. Assessing and restructuring are both based on the notion of cohesion of a class. Less cohesive classes can be transformed into a form of an abstraction by restructuring them.

1. Introduction

Object-orientation aims to model the real world as closely to a user's perspective as possible. Classes play an essential role in the object-oriented development. Entities in an application domain are captured as classes, and applications are built by composing the instances which are created from them. A class consists of instance variables and methods. The instance variables of a class represent the state of instances of the class, and the methods of a class define the behaviors of its instances.

Classes serve as a unit of encapsulation; that is, instances of a class can be manipulated only through the interface defined in the class. Therefore, the internal representation of classes can be changed without affecting any clients as long as the new representation conforms to the same (or an upward compatible) interface. Classes thus assure designers that compatible changes can be made safely, which facilitates program evolution and maintenance[7].

In order to take the advantage provided by classes, such as data abstraction and encapsulation, real-world concepts should be modeled properly as classes[6]. Classes which do not model an entity appropriately can be a serious obstacle to the development of systems because object-oriented systems are often developed by reusing the existing classes. Therefore, it is required to assess the quality of classes and transform the classes of poor quality into better ones.

This paper proposes a method for assessing the quality of classes based on cohesion and a method for restructuring them. We adopt cohesion for assessing the quality of classes. A class is said to be highly cohesive when its members, instance variables and methods, have tight relatedness among themselves. If a class is a model of an entity in an application domain, then the members of the class will have tight relatedness among themselves, which leads to a highly cohesive class.

Some researches have been conducted to define the cohesion for classes or abstract data types(ADTs), but they have some weakness. First, the previous works[1, 3] do not take into account the role of accessor method which references only one instance variable in the class and whose only behavior is to retrieve or update its value. Encapsulation of classes promotes the use of such a simple method and the reference of one instance variable is sufficient for the method to complete its behavior. Therefore, it does not make sense that accessor method weakens the cohesion of the class. Second, the previous works consider only the number of the interactions, not the pattern of the interactions among the members of a class. Briand[1] defines cohesion for abstract data types(ADTs) as the ratio of actual interactions to all possible interactions. Figures 1 (a) and 1 (b) show the interaction graphs of modules A and B, respectively. According to this definition, these two modules have the identical cohesion[1] despite the distinct patterns of the interactions: the interaction graph of module A is connected, but that of module B

[1]The number of all possible interactions is 12, and the number of interactions taking place actually is 6; consequently, the cohesion of modules A and B is 6/12.

0-8186-7638-8/96 $05.00

is disjoint. From the definition of cohesion(i.e. relatedness among the elements of a module), module A should be considered more cohesive than module B.

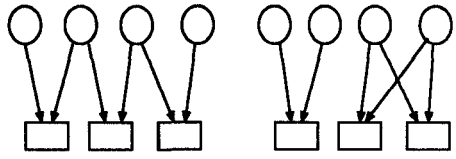

Figure 1. An example for Briand's cohesion

We propose a new cohesion which considers the role of accessor methods and the interaction patterns. We define the most cohesive form of a class, and cohesion of a class is defined as the extent to which the class approach the most cohesive form. The most cohesive form of a class is not only a basic concept to the definition of cohesion, but also to the restructuring method. That is, we propose a method for restructuring a less cohesive class by which it is transformed into the most cohesive form.

The remainder of this paper is organized as follows. Section 2 defines a new cohesion and presents a method of evaluating the quality of classes based on it. Section 3 discusses a method of restructuring less cohesive classes, and an example is presented in Section 4. Conclusion and future works are given in Section 5.

2. Assessing the quality of Classes

In this paper, we use the *reference graph*[2] to represent the members of a class and the interactions among them.

Definition 2.1 *The* **reference graph** *for class C, denoted by $RG(C)$, is a directed graph $G = (N, A)$ with vertices $N = V(C) \cup M(C)$ and edges $A = \{(m_i, v_j) \mid m_i$ references v_j, $m_i \in M(C)$, $v_j \in V(C)\}$, where $V(C)$ is a set of instance variables in class C, and $M(C)$ is a set of methods in class C. A method m_i is said to reference an instance variable v_j if m_i uses or defines the value of v_j.*

Figure 2 depicts a reference graph for a class *Stack*. The variables *top* and *store* are instance variables, and *is_empty*, *push*, and *pop* are methods. The method *is_empty* references only *top*, and the methods *push* and *pop* reference both *top* and *store*.

The cohesion for a class is evaluated in terms of the degree of the interactions between the methods and the instance variables. We claim that the cohesion should be proportional to the binding strength among

[2]Embley[4] defines reference graph for representing an ADT.

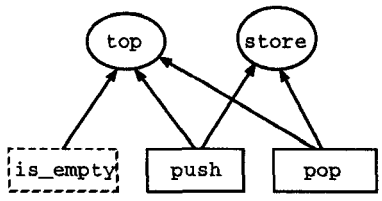

Figure 2. A reference graph of a class *Stack*

the members of a class: the more strongly bound are its members, the more cohesive the class is. For example, Figures 3 (a), 3 (b), and 3 (c) show the reference graphs for classes A, B, and C, respectively. Those reference graphs can be made disjoint by removing zero, one, and two method(s), respectively: Class B can be disjoint by removing either M_2 or M_3, and class C becomes disjoint by removing both M_2 and M_3. In other words, the members of classes A, B, and C are bound by zero, one, and two method(s). Therefore, it can be claimed that class C is more cohesive than class B and that class B is more cohesive than class A. We define a new cohesion for a class based on the observation that a class whose members are bound by a larger number of methods is more cohesive.

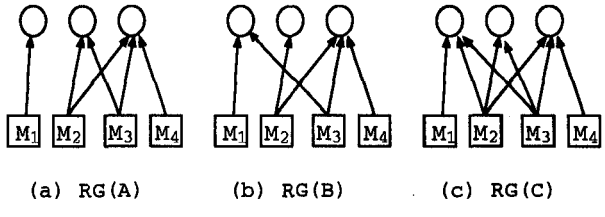

Figure 3. Classes which show distinct binding strength

2.1 Basic Definitions

Several underlying concepts precede the definition of cohesion for classes.

Definition 2.2 *A method in class* C *is a* **single** *method if it references only one instance variable in class* C.

Definition 2.3 *A single method is* **simple** *if its only behavior is to send one message to the instance variable. A single method is* **not simple** *if it performs several actual computations without simply sending one message to the instance variable.*

The method *is_empty* in Figure 4 is a simple method whose only behavior is to send a message *is_zero* to the

instance variable *top*. A simple method is depicted as a rectangle with a dotted line in the reference graph. The method *search* in Figure 5 is single, but not simple; that is, the method *search* references only the instance variable *array*, but performs several operations such as iteration and branch.

```
int Stack::is_empty() {
  return top.is_zero() ;
}
```

Figure 4. An example of a simple method

```
int SomeClass::search(int value) {
  for ( int i = 0 ; i < array.get_size() ; i ++ )
    if ( array.get_value(i) == value ) return i ;
  return -1 ;
}
```

Figure 5. An example of a non-simple method

Definition 2.4 *An* **abstraction**[3] *is the most cohesive and desirable form of a class. The following conditions should be satisfied for a class* C *to become an abstraction.* $M_s(C)$ *denotes the set of single methods in* C.

1. $\forall m \in M_s(C)$, *m is a simple method.*
2. $\forall m \notin M_s(C), \forall v \in V(C)$, *m has a arc to v.*

Informally speaking, a class is considered to be an abstraction if each single method in the class is a simple method and each non-single method references all the instance variables in the class. For example, the class in Figure 6 (a) is not an abstraction because the method M_2, a non-single method, does not reference the instance variable V_3. The class in Figure 6 (b) is not an abstraction because the method M_4, a single method, is not simple. The class in Figure 6 (c) is an abstraction because the methods M_1 and M_4 are simple as well as single, and the non-single methods M_2 and M_3 reference all of the instance variables V_1, V_2, and V_3.

[3]Ideally, a class should be an abstraction of an entity in real world.

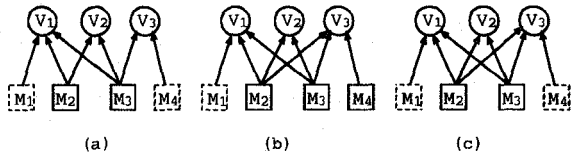

Figure 6. Examples for abstraction

2.2 Cohesion for a class

A *structure tree* for a class, constructed from the reference graph of the class, represents the structural characteristics of the class, and we define the cohesion of a class using its structure tree.

The structure tree for a class is constructed by decomposing its reference graph recursively until each partitioned sub-reference graph has a sufficiently strong cohesion. First, we identify a minimum set of methods[4] which can separate the reference graph of the class. We define a minimum set of methods which can separate the reference graph as the *glue methods*. By removing the glue methods, each of the partitioned sub-reference graphs can compose a child node.

This decomposition procedure is repeated recursively to each of the children until each of the methods references either only one or all the instance variables in the sub-reference graph.

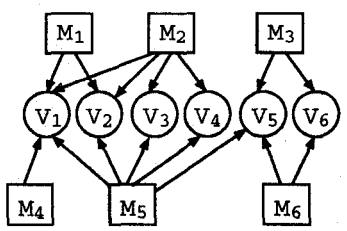

Figure 7. Reference graph of class A

Definition 2.5 *The* **structure tree** *for a class C, denoted* $ST(C)$*, is a tree whose nodes are* (rg, M_g)*, where* $rg \subseteq RG(C)$*, and* $M_g \subseteq M(rg)$ *and* M_g *is the glue methods of* rg*. The root node of* $ST(C)$ *is* $(RG(C), M_g(RG(C)))$*. A node* (rg_c, M_c) *is a child of a node* (rg_p, M_p) *when* rg_c *is one of the sub-reference graphs which are obtained from* rg_p *by removing the methods* M_p*.*

For example, Figure 8 shows a structure tree for class A whose reference graph is shown in Figure 7. Class A has instance variables V_1, V_2, ..., V_6 and

[4]In some case, a reference graph can be decomposed by two or more minimum sets of methods. This is discussed in Section 5

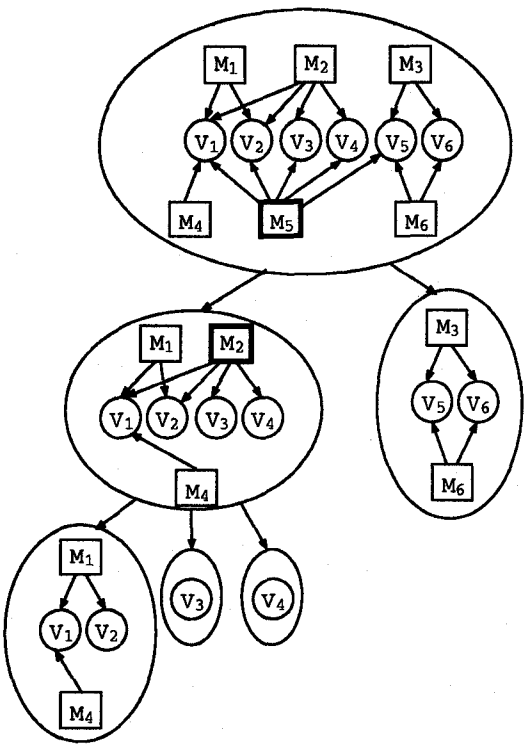

Figure 8. A structure tree of class A

Definition 2.7 *The* **cohesion** *of a class C, denoted by $CO(C)$ is defined as the tuple $(BS(RG(C)), CO(RG_1), CO(RG_2), \ldots, CO(RG_n))$, where the RG_i, $1 \leq i \leq n$, denotes each reference graph of the children of C in $ST(C)$.*

C_1 is more cohesive than C_2 when $BS(RG(C_1)) > BS(RG(C_2))$, or when $BS(RG(C_1)) = BS(RG(C_2))$ and the children of C_1 are more cohesive than C_2. Otherwise, C_2 is more cohesive than C_1.

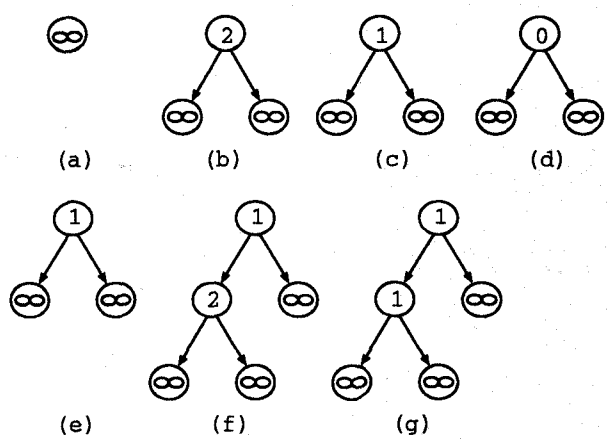

Figure 9. Examples for class cohesion

methods $M_1, M_2, \ldots, M_6$. As shown in Figure 8, the reference graph can be divided into two components by removing the method M_5; that is, the method M_5 is the glue method of the initial reference graph. A glue method is represented in rectangle with thick line in Figure 8. The right child node cannot be decomposed further because the methods M_3 and M_6 reference both the instance variables V_5 and V_6 in that component. The left child node can be partitioned into three components by eliminating the method M_2. The three components do not need to be decomposed further since each of them satisfies the terminating condition mentioned above.

The reference graph of a leaf node exhibits very strong cohesion and is close to an abstraction except that the leaf node can contain non-simple methods. The non-simple methods at a leaf node can be transformed into simple methods by restructuring methods discussed in Section 3. Consequently, it can be said that a structure tree represents how a less cohesive class is decomposed into a collection of abstractions.

Definition 2.6 *The* **binding strength(BS)** *indicates the strength of interconnectivity of a reference graph. Binding strength of a decomposable reference graph is defined as the number of its glue methods. A reference graph which cannot be decomposed further is defined to have infinite(∞) binding strength.*

A component is more cohesive when it has a greater binding strength or the same binding strength with more cohesive children. Consider the structure trees in Figure 9, where a circle represents a node of the structure tree, and the binding strength is shown in the circle. While the class in Figure 9 (a) is not decomposable, each of the classes in Figures 9 (b), 9 (c), and 9 (d) can be decomposed into two components by removing two, one, and zero method(s), respectively. Hence, the class in Figure 9 (a) is more cohesive than Figures 9 (b), Figure 9 (b) more cohesive than Figure 9 (c), and Figure 9 (c) more cohesive than Figure 9 (d) because the former reference graph has a greater binding strength than the latter reference graph.

All of the classes (e), (f), and (g) in Figure 9 can be partitioned into two components by removing one method. That is, their root nodes have the same binding strength 2. However, both children of (e) have infinite binding strength, but one child of (f) and (g) have binding strength 2 and 1, respectively. Therefore, we conclude that (e) is more cohesive than (f), and (f) more cohesive than (g). Consequently, the classes in decreasing order of cohesion are (a), (b), (c), (f), (g), and (d) (The class (e) has the same cohesion as (c).).

When decomposing a reference graph, there is not always a unique way to decomposing the reference graph. That is, a reference graph can be separated by two or more sets of glue methods. In that case,

we decompose the reference graph by choosing a set of glue methods which results in the higher cohesion of the children.

3. Restructuring of Classes

Restructuring of a class can be applied to a relatively less cohesive class to improve its cohesion. We propose two restructuring methods: creation of a component class for a highly cohesive child node, and transformation of non-simple methods into simple methods. These two restructuring methods transform a less cohesive class into a form of an abstraction.

3.1. Creation of a new class for a child node

A component in the child node can be a candidate for a new class. When a child node has a strong cohesion and can be considered to be a model of an entity, we can create a new class for the component and replace it with an instance of the new class.

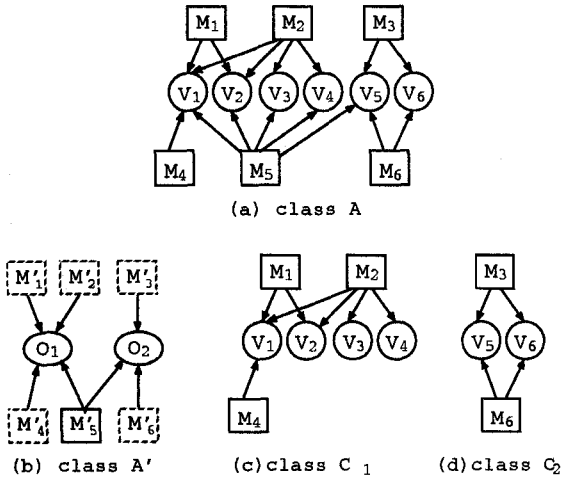

Figure 10. Creation of a new class for a child node

Figure 10 depicts the restructuring procedure for the class A shown in Figure 7. As shown in its structure tree in Figure 8, two components are identified by removing the glue method M_5. One has the methods M_1, M_2, and M_4, and the other has the methods M_3 and M_6. For each component, we create classes C_1 and C_2 respectively(see Figures 10 (c) and 10 (d)). Figure 10 (b) shows the class A' restructured from class A. It has instance variables O_1 and O_2 which are the instances of classes C_1 and C_2, respectively. The method M'_1 in the class A' conducts the same behavior as the method M_1 in the class A by invoking the method M_1 in O_1. This is the same for the methods M'_2, M'_3, M'_4, and M'_6. That is, those methods are simple methods which invoke the corresponding methods in the component objects. Implementation of M'_5 is the same as that of M_5 except that the direct reference to the instance variable is replaced with the invocation of the corresponding accessor method. Accessor methods for each instance variable should be provided in C_1 and C_2 because the method M'_5 can no longer access the instance variables in O_1 and O_2. For convenience, the accessor method is not depicted in Figures 10 (c) and 10 (d).

As shown in Figure 10, class A is transformed into A', a form of an abstraction; each of the methods in class A' either references all of the instance variables or is a simple method.

3.2. Transformation of non-simple methods into simple methods

A new method whose behavior corresponds to a non-simple method is created for the class of the object that the non-simple method accesses, and then several operations of the non-simple method is transformed into a single invocation to the newly defined method.

For example, the method *SomeClass::search* in Figure 5 can be transformed into a simple method. We first define a new method *Array::search*, which shows the idenitical behavior with the method *SomeClass::search*, for class *Array*[5] and then have *SomeClass::search* invoke the newly introduced method *Array::search*.

4. Example

This section examplifies the proposed restructuring methods. Class *ErrorDialog_1*, shown in Figure 12, a wrapper class for *Error Dialog Box* widget in Motif, and its reference graph is depicted in Figure 11. Class *Errordialog_1* is not an abstraction because the non-single method such as *activate* and *ok* does not reference all the instance variable, and the single methods such as *set_message* and *set_title* are not simple. Its reference graph can be decomposed into two components by removing the glue method *activate*, and the children are not decomposed further.

We can restructure class *ErrorDialog_1* by creating two classes which correspond to the children. Class *DialogWidget* (Figure 13 (c)) is created for the child which has methods *set_message* and *set_title* and instance variable *dialog_widget*. Class *UserHandler* (Figure 13 (b)) is created for the other methods and instance variables. The restructrued class *ErrorDialog_2* (Figure 13 (a)) has two instance variables which are instantiated

[5]Instance variable *array* is assumed to an instance of class *Array*

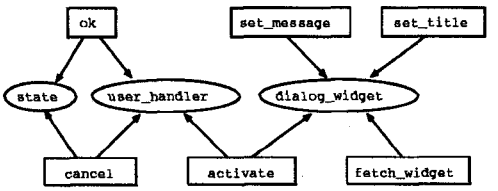

Figure 11. Reference graph for *ErrorDialog_1*

from two classes and preserves the interface of class *ErrorDialog_1*. The behavior of the methods in *ErrorDialog* such as *ok*, *cancel*, *set_message*, and *set_title* is preserved by invoking the corresponding method in classes *DialogWidget* or *UserHandler*. The direct reference to the instance variables in method *activate* is converted to the invocation of the corresponding accessor method in classes *DialogWidget* and *UserHandler*. As shown in the reference graph in Figure 13, class *ErrorDialog_2* is a form of an abstraction; each of the methods *ok*, *cancel*, *set_message*, *set_title*, and *fetch_widget* is a simple method, and the method *activate* references all of the instance variables *dialog_widget* and *user_handler*. Class *ErrorDialog_2* is shown in Figure 14.

5. Conclusion

In this paper, we define a new cohesion for evaluating the quality of classes. An abstraction is proposed as the most cohesive form of a class; we believe that when each of the methods in a class either references all the instance variables or just sends one message to one instance variable, the class is most cohesive. Binding strength has been proposed for indicating how strongly the members of a class are bound. Our proposed cohesion is based on the binding strength of a class; the greater binding strength a class has, more cohesive the class is. Cohesion of a class is defined by applying recursively the concept of binding strength to its structure tree, which is obtained by decomposing the reference graph until each of the sub-reference graphs is similar to an abstraction except that it can have a single method; when the binding strengths of two reference graphs are same, the one which has children of greater binding strength is more cohesive.

Our approach to cohesion reflects the role of accessor method and the interaction patterns; accessor methods does not weaken the cohesion and the factor which influences the cohesion of a class is not the number of interactions, but their pattern. Even the cohesion does not give a quantatative measure to a single class, it enables us to compare cohesion between classes.

In order to improve the quality of classes, we have also proposed two restructuring methods which are based on the notion of cohesion. Two restructuring methods can transform a less cohesive class into the

```
# include <Xm/Xm.h>
typedef void (*ErrorDialogHandler)(int state) ;
# define OK 0
# define CANCEL 1
class ErrorDialog_1 {
  private:
  Widget dialog_widget ;
  int state ; // OK, CANCEL
  ErrorDialogHandler user_handler ;
  void fetch_widget () {
    // fetch widget from uid file and realize it.
    // this function references dialog_widget
  }
  public:
  void set_title (char* str) {
    XmString compound_string ;
    compound_string = XmStringCreate(str,
      XmSTRING_DEFAULT_CHARSET) ;
    XtVaSetValues (dialog_widget, XmNdialogTitle,
      compound_string, NULL) ;
    XmStringFree(compound_string) ;
  }
  void set_message (char* msg) {
    XmString compound_string ;
    compound_string = XmStringCreate(msg,
      XmSTRING_DEFAULT_CHARSET) ;
    XtVaSetValues (dialog_widget, XmNmessageString,
      compound_string, NULL) ;
    XmStringFree(compound_string) ;
  }
  void activate (char* title, char* msg,
    ErrorDialogHandler handler) {
    fetch_widget() ;
    user_handler = handler ;
    set_title(title) ;
    set_message(msg) ;
    XtManageChild(dialog_widget) ;
  }
  void ok () {
    state = OK ;
    if ( user_handler) user_handler(state) ;
  }
  void cancel () {
    state = CANCEL ;
    if ( user_handler ) user_handler(state) ;
  }
} ;
```

Figure 12. Class *ErrorDialog_1*

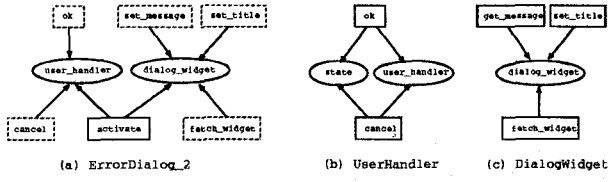

Figure 13. Reference graph for *ErrorDialog_2*

most cohesive and desirable form of a class, an abstraction.

With the assistance of development environment, the proposed methods can help evaluate and improve the quality of classes, thereby increasing the maintainability and reusability of object-oriented programs. A validation of the proposed methods by experiments remains as future works.

References

[1] Lionel Briand et al., "Defining and Validating High-Level Design Metrics," *Technical Report CS-TR-3301-1*, University of Maryland, 1994

[2] Eduardo Casais, "Managing Evolution in Object Oriented Environments: An Algorithmic Approach," *PhD thesis*, University of Geneva, 1991

[3] Shyam. R. Chidamber and Chris F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. on Software Engineering*, pp. 476-493, Jun. 1994

[4] David W. Embley and Scott N. Woodfield, "Assessing the Quality of Abstract Data Types Written in Ada," *Proc. of 10th Int. Conf. on Software Engineering*, 1988

[5] Tom
Korson and John D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Comm. of the ACM*, pp. 40-60, Sep. 1990

[6] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, 1991.

[7] Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proc. of 1th ACM Conf. on Object-Oriented Systems, Languages, and Applications*, Sep. 1986


```
class UserHandler {
  private:
  ErrorDialogHandler user_handler ;
  int state ;
  public:
  void set_handler(ErrorDialogHandler handler) {
    // additional accessor method
    user_handler = handler ;
  }
  void ok () { // same as ErrorDialog_1::ok }
  void cancel () { // same as ErrorDialog_1::cancel }
} ;
class DialogWidget {
  private:
  Widget dialog_widget ;
  public:
  Widget get_widget() { // additional accessor method
    return dialog_widget ;
  }
  void fetch_widget () {
    // this method is exported so that ErrorDialog_2
    // can access it.
    // same as ErrorDialog_1::fetch_widget
  }
  void set_title (char* str) {
    // same as ErrorDialog_1::set_title
  }
  void set_message (char* msg) {
    // same as ErrorDialog_1::set_message
  }
} ;
class ErrorDialog_2 {
  private:
  UserHandler user_handler ;
  DialogWidget dialog_widget ;
  public:
  void activate (char* title, char* msg,
    ErrorDialogHandler handler) {
    dialog_widget.fetch_widget() ;
    user_handler.set_handler(handler) ;
    dialog_widget.set_title(title) ;
    dialog_widget.set_message(msg) ;
    XtManageChild(dialog_widget.get_widget()) ;
  }
  void ok() { user_handler.ok() ; }
  void cancel() { user_handler.cancel() ; }
  void set_title(char* title) {
    dialog_widget.set_title(title) ;
  }
  void set_message(char* msg) {
    dialog_widget.set_message(msg) ;
  }
} ;
```

Figure 14. Class *ErrorDialog_2*