

# Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs

David C. Kung, Jerry Gao, Pei Hsia  
Department of Computer Science and Engineering  
The University of Texas at Arlington  
P. O. Box 19015, Arlington, TX 76019-0015  
Telephone: (817) 273-3627  
Fax: (817) 273-3784  
Email: *kung|gao|hsia@cse.uta.edu*

Jeremy Lin, Yasufumi Toyoshima  
Fujitsu Network Transmission Systems, Inc.

October 19, 1993

**Abstract:** Regression testing is an important activity in software maintenance. Although a number of existing research results have addressed the problems and solutions in regression testing of traditional programs, no research results have focused on the issues and solutions in regression testing of object-oriented programs. In this paper, we present a method for identifying the affected classes when changes are made to an object-oriented program. An algorithm for generating a desirable order to test the affected classes also is described. The basic model we use is an object relation graph, which depicts the inheritance, aggregation, and association relations that exist in the object-oriented program to be maintained. The test order finding algorithm can be applied to acyclic as well as cyclic object relation graphs. The results of this paper has been implemented and applied to testing of many example applications, including the InterViews library.

## 1 Introduction

Regression testing involves re-testing part of a software system after it is modified. The modification may be caused by specification, or code change. The objective of regression testing

is to ensure that the modified program still satisfies its requirements. To save effort and time, regression testing needs only re-test those parts that are affected by the modification.

Regression testing has to address four fundamental problems: 1) how to automatically identify the affected components due to changes of some components; 2) what strategy should be used to retest these affected components; 3) what are the coverage criteria for re-testing these components; and 4) how to select, reuse, and modify the existing test cases (and generate new ones).

Solutions to these problems for traditional programs have been proposed during the last two decades (e.g., [Fischer 1981] [Harrold 1988] [Harrold 1989] [Laski 1992] [Leung 1990] [Prather 1987] [White 1992]). However, testing of OO programs has received very little attention; regression testing of OO programs has received almost none.

The object-oriented paradigm for software development introduces a number of new concepts, such as class, inheritance, encapsulation, dynamic binding, and polymorphism. These new concepts result in complex relationships between classes and their members. They not only introduce new testing problems as recognized in [Harrold 1992] [Perry 1990] [Smith 1990] [Wilde 1991], but also raise a new challenging question of how to conduct regression testing for object-oriented programs.

Although the existing results can be applied to regression testing of member functions of a class at the unit and integration levels, they are not suitable for testing components at higher levels, such as a class, a group of classes, or class libraries. First, traditional approaches do not address the complex relationships and dependencies, such as inheritance, aggregation, and association, that exist between classes. Second, most traditional approaches are based on the control flow model, but class objects have state dependent behavior that can change in various ways; and hence, traditional approaches cannot be applied to class testing. Third, traditional approaches use test stubs to simulate the modules that are invoked but in OO programs this is difficult and costly because it requires understanding of many related classes, their member functions, and how the member functions are invoked [Smith 1990] [Wilde 1992].

In this paper, we propose a method for regression testing of OO programs. The underlying process of the method consists of four activities: 1) change analysis to identify affected components; 2) retest strategy generation to produce a cost-effective class test order; 3) test case reuse, modification, and generation; and 4) regression test plan implementation to retest the modified program. This paper only provides detailed technical solutions for problems in the first two activities.

The first two activities are performed using a class regression test model, called the object relation graph (ORG). It describes the inheritance, aggregation and association relations and captures the dependencies between the classes. We define a class firewall to enclose all the classes affected by changes to one or more classes. The algorithm for computing the firewall is transitively derived using the dependencies. We introduce a new, cost-effective test strategy (called test order), which is derived also from the dependencies, to provide the tester with a detailed road map for conducting retests of classes in the firewall.

The organization of this paper is as follows. Section 2 gives a brief review of previous solutions to the four fundamental problems in regression testing. In section 3, we present an object relation graph (ORG) as a test model for class level regression testing. In section 4, the concept of a class firewall is introduced to enclose all affected classes when changes are made to one or more classes. Moreover, a method for constructing class firewalls is also described. Section 5 is devoted to test strategy generation, i.e., finding the most cost-effective test order for testing the classes in a class firewall. Application of our results to the InterViews class library is presented in section 6. In section 7, we give conclusions and future work.

## 2 Related Work

In this section, we briefly review existing work on regression testing, discuss problems, and relate our work to the existing ones.

General discussions on regression testing can be found in [Leung 1989] [Hartmann 1988]. Existing solutions to the four fundamental problems described in section 1 are discussed in the following paragraphs.

The first problem is automatic identification of affected modules or parts. Harrold *et al.* [Harrold 1988] introduced a technique for analyzing the change effects within a module. The idea is using a data-flow graph to identify the affected definition-use pairs and/or subpaths. The advantage of this approach is that test effort is reduced by re-testing only the affected define-use paths and new paths. The technique was later extended so that it can also be used to identify affected procedures at the interprocedural level [Harrold 1989]. Unlike Harrold's technique, a number of researchers proposed different methods based on a control-flow graph of a procedure/function to identify the affected control paths in a module [Laski 1992] [Benedusi 1988] [Prather 1987].

At the module integration level, H. K. J. Leung and L. White [Leung 1990] introduced the firewall concept to enclose the affected modules due to a module modification. The notion of a control related firewall is defined based on a call graph. Effort is reduced by re-testing only the modules and links in the firewall of the changed module. The notion of a data related firewall was also introduced by the same authors [White 1992]. It is based on a data-flow graph to enclose all affected modules due to coupling through global data.

The second problem of regression testing is finding a cost-effective test sequence for conducting retests so that test effort and costs are minimized. The well-known test strategies include the top-down, bottom-up, and sandwich approaches [Beizer 1990]. These approaches rely on the tester to make the selection. Prather et al [Prather 1987] proposed an adaptive path prefix software testing strategy that utilized previous test paths as a guide in the selection of subsequent paths. Their method ensures branch coverage and consumes fewer computational resources. Harrold et al [Harrold 1992] presented an incremental OO testing methodology based on class inheritance hierarchy. The approach suggests that base class should be tested before derived classes so that the base class' test cases and relevant information can be reused

in testing the derived classes.

The third problem of regression testing concerns coverage criteria. Fischer, and Prather et al, respectively, described the various retest criteria relating to path coverage of a function/procedure [Fischer 1977] [Fischer 1981] [Prather 1987]. Leung and White used firewalls as a re-testing criteria at module level to ensure that all affected modules and links between modules will be retested [Leung 1990] [White 1992].

The last problem relates to how to the selection, reuse, and modification of existing test cases for re-testing. Fischer described a test case selection strategy which takes the form of a set covering problem [Fischer 1977] [Fischer 1981]. The basic idea is using the concept of 0-1 integer programming models to find the minimum test cases which covers one of the path criteria in unit regression testing. Lee and He also used the 0-1 integer programming model on a test matrix to minimize test efforts in functional regression testing [Lee 1990]. Leung and White proposed a *Retest Strategy* for performing corrective regression testing in [Leung 1989]. The main idea is to view regression testing as composed of two subproblems: the test selection problem and the test plan update problem. Thus, the re-testing process is divided into two phases: test classification and test plan update. After the existing test cases are classified into: reusable tests, obsolete tests and retestable tests in the test classification phase, only retestable test cases and new test cases are considered as tests in the new regression test plan.

The existing methods can be applied to regression testing of member functions [Fiedler 1989]. Traditionally, testing uses test stubs and drivers to simulate the called functions and calling functions. In OO testing, this is both costly and difficult, because the tester has to understand a chain of member functions and classes before he can construct a stub or driver.

In this paper, we only address the first two problems of regression testing of object-oriented programs; viz. identification of affected modules or parts, and finding a a cost-effective test sequence. The last two will be addressed in future publications. Our solution to identifying affected classes has been influenced by Leung and White's firewall concept for modules. However, our approach aims at identifying the affected classes instead of modules. This problem has not been addressed in the literature and its complexity, as discussed in section 1, calls for innovative solutions. As a testing strategy, we propose to find a desirable order, called class test order, to test the affected classes. The difference between our test order and that described in [Harrold 1992] is that we take into consideration not only the inheritance relationship, but also the aggregation, and association relationships. The notion of class test order and its associated algorithms as presented in this paper can be applied to not only class inheritance hierarchies, but also class libraries, and application programs.

### **3 The Class Regression Test Model**

The class regression test model, called the object relation graph (ORG), is introduced to represent the inheritance, aggregation, and association relationships between the classes. It

is used to identify the affected classes (when one or more classes are changed) and generate a cost-effective test order for testing the affected classes. The ORG is only a part of our formal test model, which describes also the control structure of the member functions and the state dependent behaviors of the objects. A detailed description of the complete formal test model can be found in [Kung 1993].

Before we proceed to describe ORG, we first briefly review the inheritance, aggregation, and association concepts used in OO modeling [Booch 1991] [Rumbaugh 1991]. The purpose is to provide the necessary background for those readers who are not familiar with the OO paradigm. We also provide simple examples to illustrate these relations in C++.

### 3.1 Class Relations

The three most widely used class relations in OO modeling are the inheritance, aggregation, and association relations [Booch 1991] [Rumbaugh 1991] [Coad 1990] [Korson 1990]. Inheritance means properties defined for an object class are automatically defined for all of its subclasses (unless selective and/or overriding inheritance are specified). Aggregation means an object class is “*part-of*” or “*contained in*” another object class. Association means some kind of connection between two classes. For example, a person owns a car can be considered an association between a Person instance and a Car instance.

C++ provides direct support for inheritance. For example, Car is a subclass of Vehicle can be defined as

```
class Vehicle { ... };
class Car: public Vehicle { ... };
```

where “{ ... }” denotes the code that defines the data members and function members that are specific to the class. The definition means 1) Car inherits the type defined for Vehicle; and 2) all the public accessible data members and function members of Vehicle are also public accessible members of Car. Other access controls such as *private* and *protected* are also supported by C++ but it is beyond the scope of this paper to present these. The reader is referred to [Stroustrup 1991] [Lippman 1991].

Aggregation can be implemented in C++ using member data. For example, Car has one engine, four tires, and other components as its parts can be defined as follows:

```
class Engine { ... };
class Tire { ... };
class Car {
  Engine e; // car has an engine
  Tire t[4]; // car has four tires
  // other components
};
```

Another way of implementing aggregation is to have the aggregate object dynamically create

the component objects. For example, a book contains chapters, some of which may be unknown before the book is completed:

```
class Chapter { // define a Chapter object class
char* title;
Chapter* next; // pointer to next chapter, if any
// other data members
// ...
};

class Book {
char* title; // title of book
Chapter* first_chapter, last_chapter; // a book consists of a list of Chapters
// ...
};

void Book::add_chapter (char* title, //...) { //add a new chapter
Chapter *chap = new Chapter; // dynamically creates a new chapter
chap.title = title; // fills in chapter information
//...
last_chapter.next = chap; // appends the chapter to the book
chap → next = 0; last_chapter = chap; }
```

In this example, the Chapters are created by Book, they are part-of Book, not independent objects.

Unlike aggregation, association represents more general relationships or connections between objects. Either of the related objects is not a part of the other. The objects relate themselves through access to others objects' data, or through message passing. Consider, for example, the Person *owns* Car association. In this, neither Person nor Car is part of the other. One way of implementing this is:

```
class Car { ... };
class Person {
Car *mycar; // relates person to car through a pointer
// ...
... };

void Person::add_ownership (Car *p) {
// ...
p → next = mycar; mycar = p;
}
```

In this case, Car is not part of Person and it may also be owned by the person's spouse.

It should be pointed out that these examples are not meant to exhaust all cases. However,

the regression test model described in the next section does take into consideration all the possible combinations.

In summary, the inheritance relation can be easily extracted because these are explicitly defined in C++. The aggregation relation involves code that includes other object class as part of the aggregate class, or code that dynamically creates objects of the component classes. In the case of association, the related object classes are not part of one another, and the related objects not created by the object. These are the principles that we use in the development of the regression test model.

### 3.2 Notion of Object Relation Graph

An object-relation graph, called ORG, is defined below to capture the relationships between different classes and their objects.

**Definition 1.** An edge labeled digraph  $G = (V, L, E)$  is a directed graph, where  $V = \{V_1, \dots, V_n\}$  is a finite set of nodes,  $L = \{L_1, \dots, L_k\}$  is a finite set of labels, and  $E \subseteq V \times V \times L$  is the set of labeled edges.

**Definition 2.** The ORG for an OO program P is an edge labeled directed graph (or digraph in short)  $ORG = (V, L, E)$ , where  $V$  is the set of nodes representing the object classes in P,  $L = \{I, Ag, As\}$  is the set of edge labels, and  $E = E_I \cup E_{AG} \cup E_{AS}$  is the set of edges defined below.

**Definition 3.**  $E_I \subseteq V \times V \times L$  is the set of directed edges representing the inheritance relation between the classes. For any two classes  $C_1, C_2 \in V$ ,  $\langle C_1, C_2, I \rangle \in E_I$  if and only if one of the following declarations appears in the header files of P:

- “class  $C_1: C_2$ ” or “class  $C_1: \text{private } C_2$ ”.
- “class  $C_1: \text{public } C_2$ ”.
- “class  $C_1: \text{protected } C_2$ ”.

**Definition 4.**  $E_{AG} \subseteq V \times V \times L$  is the set of directed edges representing the aggregation relation between the classes. For any two classes  $C_1, C_2 \in V$ ,  $\langle C_1, C_2, Ag \rangle \in E_{AG}$  iff one of the following declarations appears in the source files of the program P:

- class  $C_1$  {  
 $C_2$  b; // an instance of  $C_2$  is part of  $C_1$   
 $C_2$  c [m]; // an array of instances of  $C_2$  is part of  $C_1$   
 ... };

appears in the header files of the program P. This is called *automatic aggregation*.

- class  $C_1$  {...
  - static  $C_2$  b; // a static instance of  $C_2$  is part of  $C_1$
  - static  $C_2$  c [m]; // a static array of instances of  $C_2$  is part of  $C_1$
  - ... };

appears in the header files of the program P. This is called *static aggregation*.

- class  $C_1$  {...
  - $C_2$  \*b; // a dynamically created instance of  $C_2$  is part of  $C_1$
  - static *Class\_C* \*c [m]; // an array of dynamically created instances of  $C_2$  is part of  $C_1$
  - $C_1$  (); // constructor for  $C_1$  ...};

appears in the header files of the program P. This is called *dynamic aggregation*, and

```
 $C_1::C_1$  () {
  // dynamically creating b;
  // dynamically creating c [m];
};
```

appears in some source file.

**Definition 5.**  $E_{AS} \subseteq V \times V \times L$  is the set of directed edges representing the association relation between the classes. For any two classes  $C_1, C_2 \in V$ ,  $\langle C_1, C_2, As \rangle \in E_{AS}$  iff one of the following declarations appears in the source files of the program P:

- class  $C_1$  accessed some data member of class  $C_2$ . This type of dependencies can be recognized in the following declarations:

```
(1)  $C_1::f(\dots)$  { ...
  if (b.d relational_operator data_value) {
    // b is an object of class  $C_2$ , d is a data member of b.
    // b has been defined or created outside the scope of class C1.
  }
...};
```

```
(2)  $C_1::f(\dots)$  {...
  if (p→d relational_operator data_value) {
    // p is an object of class  $C_2$ , d is a data member of b.
    p has been defined outside the scope of class.
  }
...};
```

- there is a message passing between  $C_1$  and  $C_2$ . This type of dependencies can be recognized in the following declarations:

```
(1) class  $C_1$  {
  ...
  ...f(...  $C_2$  b, ...); // b has been defined outside of the scope of class  $C_1$  and
    // passed as a parameter to  $C_1$ . ...
};
```

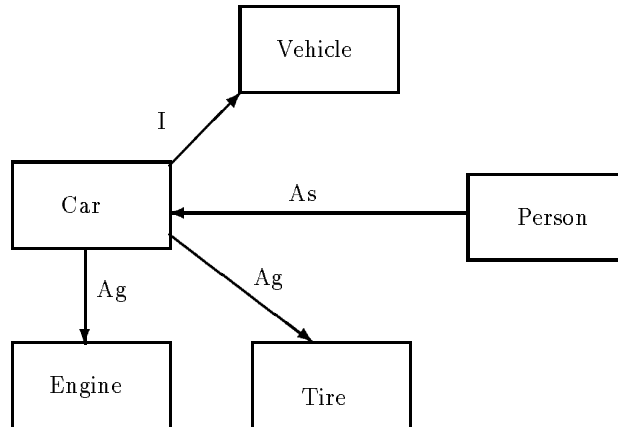


Figure 1: An ORG Example

```

(2) C1::f(...) { ...
    ...b.f(...)...; // b is an object of class C2, and is defined outside the scope of class
    C1.
    ...};
  
```

```

(3) C1::f(...) { ...
    ...p→f(...)...; // p is an object pointer of class C2,
    //the object pointed out by p has been defined outside the scope of class
    C1.
    ...};
  
```

As an example, Figure 1 depicts an ORG which represents the relationships between the classes defined in section 3.1. They are: Car, Vehicle, Engine, Tire, and Person.

## 4 Class Firewall

After a change is made in an object-oriented program, regression testing involves re-testing components at different levels: class members, classes/objects, and subsystems and systems. In order to save effort and time, regression testing needs to re-test only those components which are affected by the modification. There are various levels of changes, such as, data member change, function member change, class change, relation change. Each of these changes has different effects on the modified program. In this paper, we define class firewall as a mechanism to identify the effect of a class change. It is assumed that the relations between classes are not changed. The problem associated with class relation change will be addressed elsewhere.

## 4.1 The Notion of a Class Firewall

Intuitively, a class firewall for a class  $C$ , denoted as  $CFW(C)$ , in an OO program/library is the set of classes that could be affected by changes to  $C$ . Clearly, these classes should be retested when  $C$  is changed. We wish to point out that the notion of firewall defines the classes that are possibly be affected, not necessarily be affected.

**Lemma 1:** Let class  $A$  be a subclass of class  $B$  in the inheritance hierarchy, and only class  $B$  is changed. For adequate testing, not only should class  $B$  be unit retested, but also class  $A$  be retested with class  $B$  together if the change has effect on inherited members of class  $A$ .

**Proof:** Since class  $A$  is a subclass of class  $B$ , class  $A$  must inherit some of class  $B$ 's attributes. Thus, class  $A$  has a code dependency on class  $B$ . The modification made in class  $B$  can affect the behaviors of class  $A$ 's objects in the following two out of three cases:

- the change is made on one or more defined members of class  $B$  which are inherited by class  $A$ .
- the change is made on one or more defined members of class  $B$  which has a direct or indirect effect on inherited members of class  $A$ .
- the change has no direct or indirect effect on inherited members of class  $A$ .

Therefore, in the first two cases, class  $A$  should be unit retested to make sure that its defined members and inherited members behave correctly and interact correctly with each other. For all three cases, re-integrating test is needed for classes  $A$  and  $B$  to guarantee that all class  $A$ 's members inherited from class  $B$  properly in the reuse context.

**Lemma 2:** Let class  $A$  be a aggregate class of class  $B$  in the aggregation hierarchy, and only class  $B$  is changed. For adequate testing, not only should class  $B$  be unit retested, but also class  $A$  be retested together with class  $B$ .

**Lemma 3:** Let class  $A$  in the association hierarchy be associated to class  $B$  in one of the following two ways: (1) class  $A$  accesses class  $B$ 's data members; (2) class  $A$  has a message passing to class  $B$ . If class  $B$  is changed, for adequate testing, not only should class  $B$  be unit retested, but also class  $A$  be retested and re-integrated with class  $B$ .

## 4.2 Constructing a Class Firewall

To compute the class firewall, we first introduce a binary relation  $R$  that is derived from the directed edges of an  $ORG = (V, L, E)$ :

$$(1) \quad R = \{ \langle C_2, C_1 \rangle \mid C_1, C_2 \in V \wedge (\exists l)(l \in L \wedge \langle C_1, C_2, l \rangle \in E) \}$$

We will call  $R$  the dependence relation since it defines the dependence between the classes, according to the inheritance, aggregation, and association relations. More specifically,  $\langle$

$C_2, C_1 > \in R$  if and only if one of the following cases holds: 1)  $C_1$  is a derived class of  $C_2$ ; 2)  $C_1$  is an aggregate class of  $C_2$ , i.e.,  $C_2$  is part of  $C_1$ ; or 3)  $C_1$  is associated with  $C_2$  either by accessing its data members, or passing some messages. In all these cases,  $C_1$  is dependent on  $C_2$  in the sense that code changes to  $C_2$  would affect the behavior of  $C_1$ . The computed class firewall for a class  $C$ , denoted  $CCFW(C)$ , then is defined as

$$CCFW(C) = \{C_j \mid \langle C, C_j \rangle \in R^*\}$$

where  $R^*$  is the transitive closure of  $R$ . That is, if  $\langle C_i, C_j \rangle \in R$  and  $\langle C_j, C_k \rangle \in R^*$ , then  $\langle C_i, C_k \rangle \in R^*$ . The transitive closure of  $R$  can be computed by the famous Warshall algorithm [Aho 1983].

**Theorem 1:** Let  $G$  be an ORG of a given OO program  $P$ ,  $R$  be the dependence relation derived from  $G$ . Let  $C$  be a class in which a change is made to its defined or redefined members. Assume the dependencies between the classes of  $P$  are the dependencies of inheritance, aggregation, and association relations. Then,  $CCFW(C) = CFW(C)$ , that is

1.  $CCFW(C) \subseteq CFW(C)$ .
2.  $CFW(C) \subseteq CCFW(C)$ .

*Proof(1):* We want to prove that for any class  $C_i \in CCFW(C)$ , class  $C_i$  could be affected by the changes to class  $C$ , and should be retested. The proof is by induction on  $n$  of dependence relation  $R^n$ .

Basis: For any class  $C_i$ ,  $\langle C, C_i \rangle \in R^1$ . Then  $\langle C_i, C, l \rangle \in E$  of  $G$  based on the definition of  $R$ . Thus,  $\langle C_i, C, l \rangle$  must be one of the three cases: inheritance edge, aggregation edge, or association edge. According to Lemma 1, 2 and 3, in all these cases,  $C_i$  could be affected by changes to  $C$ , and  $C_i$  should be retested.

Inductive Hypothesis: Assume that for any class  $C_j$ ,  $\langle C, C_j \rangle \in R^k$ . Then class  $C_j$  would be affected by changes to class  $C$ , and should be retested.

Inductive Step: We need to prove that for any class  $C_i$ , if  $\langle C, C_i \rangle \in R^{k+1}$ , then class  $C_i$  could be affected and should be retested.

According to the transitive nature of  $R$ , there must be  $\langle C_j, C_i \rangle \in R$  and  $\langle C, C_j \rangle \in R^k$ . From the inductive hypothesis, class  $C_j$  could be affected by the changes to class  $C$ , and should be retested. From the basis, class  $C_i$  could be affected by the changes of class  $C_j$ , and should also be retested. Thus, class  $C_i$  could be affected by the changes to class  $C$ , and should be retested.

*Proof(2):* We want to prove that if any class  $C_i$ ,  $C_i \in CFW(C)$ , then class  $C_i \in CCFW(C)$ . We prove this using contradiction.

Assume there is class  $C_i$  which is not in  $CCFW(C)$ , but  $C_i \in CFW(C)$  in the sense that it could be affected by the changes to class  $C$  and should be retested.

Since class  $C_i$  could be affected by the change of class  $C$  only when  $C_i$  is either directly or indirectly dependent on the defined members of class  $C$ . Thus, there must be one dependency chain (denoted  $CHAIN$ ) :  $\{ \langle C_i, C_{j1} \rangle, \dots, \langle C_{jk}, C \rangle \}$ , from class  $C_i$  to class  $C$ . As we known, the types of dependency between two classes/objects can be classified into: (1) code dependency; (2) object dependency; (3) control dependency; (4) data or state dependency. According to the assumption, they are dependencies of the three relations: inheritance, aggregation, and association. Hence,  $CHAIN \subseteq R$ , and  $\langle C_i, C \rangle \in R^*$ . Therefore,  $\langle C_i, C \rangle \in CCFW(C)$ . This is a contradiction. QED.

According to theorem 1, the following theorem can be easily derived.

**Theorem 2:** Let  $CCFW(C)$  be the computed firewall for class  $C$ . For any class  $C_i$ , if  $C_i$  is not in  $CCFW(C)$ , then,  $C_i$  is not affected by the change in class  $C$  and hence, no retest is needed.

Using theorem 1, we can construct the class firewall for a changed class to enclose all possible affected classes which should be retested. According Weyuker's axioms for adequate testing[Weyuker 1986] and Perry's work[Perry 1990], these classes should also be re-integrated with their subclasses, aggregated classes and associated classes to achieve adequate testing. The detailed results for class integration test strategy is provided in the next section.

The notion of the class firewalls can be extended to a set of changed classes. Let  $S = \{C_1, C_2, \dots, C_k\}$  be a set of classes (that are changed). Then, the class firewall for  $S$ , also denoted  $CFW(S)$ , is defined as follows:

$$CFW(S) = \bigcup_{\forall C \in S} CFW(C)$$

As an example, Figure 2 depicts an ORG for a subset of classes in the InterViews library, and Figure 3 depicts the class firewall for the class Subject in the subset.

## 5 Test Order for Class Firewalls

After identifying the class firewall for a changed class, the tester needs a cost-effective strategy to conduct the retests for each class at the class unit level and re-integrate classes together at the class integration level. The existing test strategies, including top-down, bottom-up, and sandwich, rely on the tester to make the selection, and conduct the unit and integration tests by using test stubs and test drivers.

However, in regression testing of OO programs, it is costly and difficult to construct test stubs and drivers for a class object or a class function member due to the following reasons. First, simulating a class or its objects is very costly and difficult because the tester not only has to understand and simulate its class members, object states and behaviors, but also has to understand and simulate its inherited members and component class objects. Second,

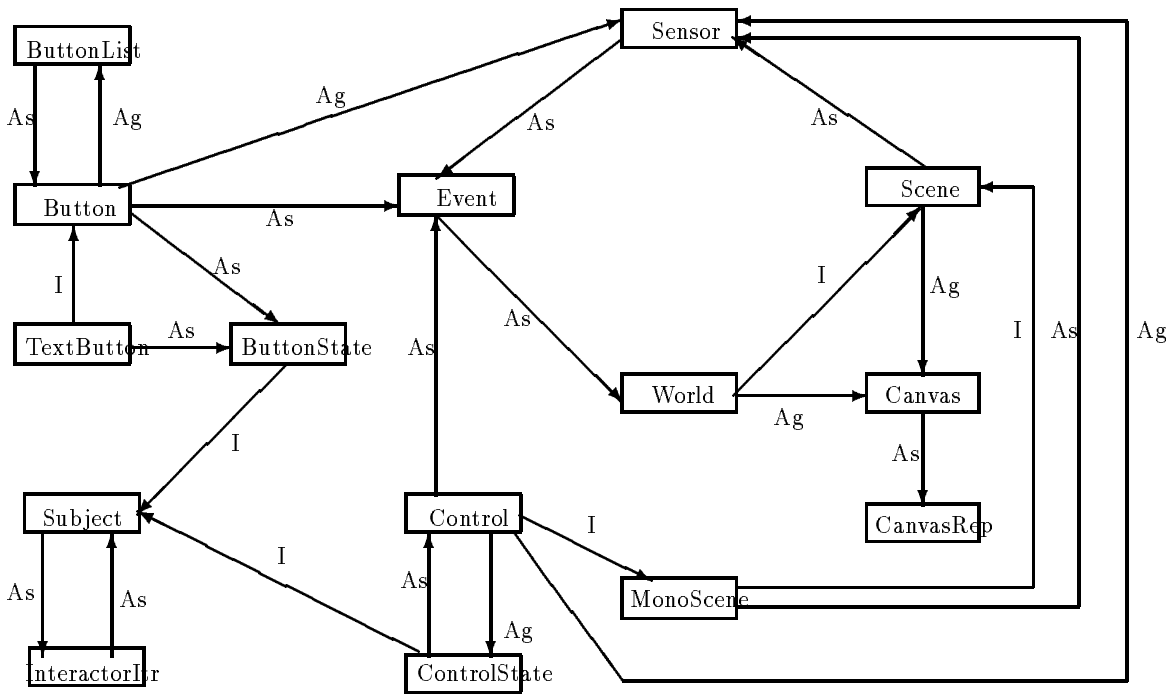


Figure 2: An ORG Example for a Subset of Classes in InterViews Library

simulating a member function is also very expensive and complicated because the test has to understand chains of member function invocations and simulate the different state changes of class objects and messages between the class objects due to these function invocations<sup>1</sup> to construct appropriate test stubs. In addition, according to [Wilde 1991], 80% member functions consist of one or two statements. Thus, if we can test components using tested components by following a proper test sequence, then the test efforts for constructing test stubs and test drivers can be reduced. The main idea is to test the independent components first, then test the dependent components based on their relationships. For example, testing the called functions before the calling function, then the effort required to construct the test stub for the calling function can be saved. Similarly, testing a base (or aggregated) class before the subclasses (or assembly classes), then the stub for the base classes (or the aggregated classes) can be reduced for testing the subclasses (or assembly classes). Moreover, the test information for the base class (or the aggregated class) can be reused in testing its subclasses (assembly classes).

In order to resolve this problem, we introduce a test strategy, called *test order*, to serve as a detailed road map in class unit re-testing and class re-integration testing.

The test order problem for class firewalls can be stated as finding a desirable order for testing

<sup>1</sup>Unlike a procedure in the conventional programs, a function member of a class can not be viewed as a function from its input domain to output domain, instead it only changes the states of class objects or passes the message between objects.

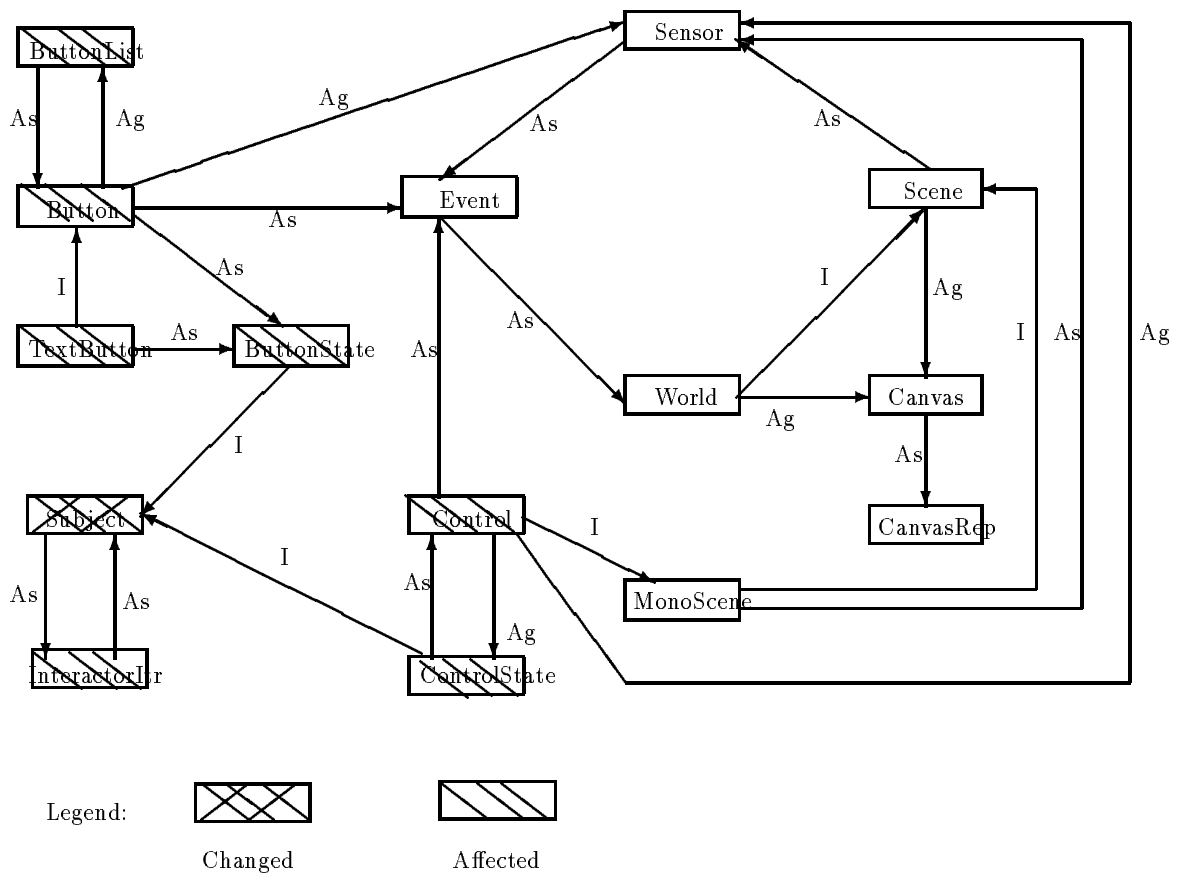


Figure 3: A Class Firewall for Class Subject in a Subset of InterViews Library

the classes that are affected by code changes to a set of classes. By testing of a class, we mean structure testing, function testing, object state testing, and/or data flow testing of the member functions of the class. Exactly how these testings are performed is beyond the scope of this paper.

A desirable test order also implies effective reuse of previously generated test cases in the new, reusing context. In [Harrold 1992], Harrold, McGregor, and Fitzpatrick proposed a methodology for reusing test cases in the testing of a class hierarchy, i.e., involving only inheritance relations. Our result on test order supplements their work and allows test cases to be reused in a more general context, where inheritance, aggregation, and association relations exist.

The above discussion implies that a desirable test order is one that requires minimum effort to construct the test stubs. Since the efforts to construct the test stubs differ substantially from case to case, we will assume that the total effort is proportional to the number of stubs need to be constructed. The reader will see later that this assumption will not affect the usefulness of the method, since it can be easily tailored to take into consideration the efforts required to construct individual stubs.

A solution to the test order problem must consider two cases:

1. The  $ORG = (V, L, E)$  is an acyclic digraph, meaning that there exists no cycle in the digraph. A cycle is a directed path leading from one node, traversing directed edges, back to itself.
2. The  $ORG = (V, L, E)$  is a cyclic digraph, meaning that there exists one or more cycles. In this case, topological sorting cannot be applied.

In the first case, the test order is simply the topological sorting of  $CCFW(S)$  using the dependence relation  $R$  (defined in section 4) as the precedence relation. The computational complexity is  $O(|CCFW(S)|)$ , i.e., the number of classes in the firewall for  $S$ , since each node needs to be visited only once [Aho 1983]. The effort required to construct the test stubs is zero and hence it is minimum.

The solution to case 2 is nontrivial since topological sorting cannot be applied to cyclic digraphs. The remainder of this paper is devoted to developing a solution for this case.

## 5.1 Overview of the test Order Finding Algorithm

The algorithm is based on two key concepts. The first is the notion of a cluster, which is a maximal set of vertices that are mutually reachable through the relation  $R$  defined in section 4<sup>2</sup>. Note that a cyclic  $ORG$  may have more than one cluster, and a cluster may contain only one vertex (such clusters are called unit clusters). A cyclic digraph  $ORG = (V, L, E)$  can be

---

<sup>2</sup>This is also called strongly connected subgraph in graph theory.

transformed into an acyclic digraph  $ORG' = (V', L', E')$  in which  $V'$  is the set of clusters in the ORG,  $E'$  is the set of edges between clusters in the ORG,  $L'$  is the set of labels on edges in  $E'$ . It can be proved that  $ORG'$  must be acyclic; otherwise, some of the clusters must not be a maximal set of mutually reachable vertices. We will not pursue a proof in this paper due to space limit. Since  $ORG'$  is acyclic, topological sorting can be applied to produce a test order for the clusters. The test order is called the major test order, to distinguish from the minor test order (described below) produced for the vertices of a cluster.

The second notion is cycle breaking, that is, to identify and temporally remove an edge(s) from a non-unit cluster so that the vertices of the cluster and their associated edges form an acyclic subgraph. Again, topological sorting can be applied to the acyclic subgraph to derive a test order, called the minor test order. Thus, a cyclic  $ORG$  can be tested first according to the major test order and then the minor test order.

We are now ready to outline the algorithm:

**Step 1.** Transform the  $ORG$  into an acyclic digraph  $ORG'$ .

**Step 2.** Produce a topological sorting for the  $ORG'$ . The test order produced is called the major test order.

**Step 3.** For each non-unit cluster of  $ORG'$  do steps 4 to 5.

**Step 4.** For each cycle of the cluster, select and remove an edges to break the cycle.

**Step 5.** Produce a topological sorting for the acyclic subdigraph obtained in step 4. The test order produced is called the minor test order.

Note the above algorithm applies to acyclic as well as cyclic digraphs. If the digraph is acyclic, then steps 3 to 5 are not performed.

Among the five steps, only steps 1 and 4 require elaboration since the other steps are straight forward.<sup>3</sup> Therefore, in the following sections, we will focus on converting a cyclic digraph to an acyclic one, and strategies for selecting an edge to break a cycle.

## 5.2 Converting a Cyclic ORG to an Acyclic ORG Using Clusters

The algorithm uses the transitive closure of the dependence relation  $R$  defined in section 3. As usual the transitive closure is denoted  $R^*$ . For any two vertices  $u, v \in V$ , where  $V$  is the vertex set of the original cyclic  $ORG = (V, L, E)$ ,  $\langle u, v \rangle \in R$  means code change to  $u$  would affect the behavior of  $v$ , and  $\langle u, v \rangle \in R^*$  means  $v$  is transitively affected by code change to  $u$ . Clearly, if  $\langle u, v \rangle \in R^*$  and  $\langle v, u \rangle \in R^*$ , then changes to  $u$  affect  $v$  and vice versa. In

---

<sup>3</sup>There are algorithms for step 1, we adapt them in the next section for readers who might not be familiar of them.

this case,  $u$  and  $v$  are mutually reachable and they are in a cycle. Therefore,  $u$  and  $v$  must be placed in the same cluster. Since the *mutually reachable* relation is an equivalence relation (i.e., reflexive, symmetric, and transitive), we follow the usual mathematical convention and denote the cluster that contains  $u$  as  $[u]$ . The purpose of this algorithm is to identify such pairs and place them in a cluster, and compute the edges that define the precedence relation between the clusters:

Input:  $ORG = (V, L, E)$

Output: an acyclic digraph  $ORG' = (V', L', E')$ , where

$$\begin{aligned} V' &\subseteq \{v' | v' \in 2^V\} \\ L' &\subseteq \{l' | l' \in 2^L\} \\ E' &= \{ \langle u', v', l' \rangle \mid (u' \times v' \times l') \cap E \neq \emptyset \} \end{aligned}$$

**Step 1. Initialize.** Let  $[v] \in V'$  for every  $v \in V$ ,  $L' = E' = \emptyset$ , and each vertex of  $V$  is marked *unclustered* (the marking will be changed to *clustered* when the vertex is placed into an established cluster). We assume that the vertices of  $ORG$  are indexed as  $v_1, v_2, \dots, v_n$ . Any indexing scheme can be used, since it is not essential.

**Step 2. Compute  $R^*$ .** Compute the transitive closure, denoted  $R^*$ , for

$$R = \{ \langle v_i, v_j \rangle \mid v_i, v_j \in V \wedge (\exists l)(l \in L \wedge \langle v_j, v_i, l \rangle \in E) \}$$

**Step 3. Compute clusters.** For  $i = 1$  to  $n - 1$ , do the following if  $v_i \in V$  is marked *unclustered*:

For  $j = i + 1$  to  $n$ , if  $\langle v_i, v_j \rangle \in R^*$  and  $\langle v_j, v_i \rangle \in R^*$ , then

1. insert  $v_j$  into the cluster containing  $v_i$ , i.e., set  $[v_i] = [v_i] \cup \{v_j\}$ ;
2. mark  $v_j$  as *clustered*; and
3. delete  $[v_j]$  from  $V'$ , i.e., set  $V' = V' - \{[v_j]\}$ .

**Step 4. Compute the edges  $E'$ .** For each pair  $u', v' \in V'$ , if there is a directed edge from some vertex in  $u'$  to some vertex in  $v'$  in the original digraph, then create a directed edge from  $u'$  to  $v'$ , the label for this directed edge is the union of all the labels of the original edges. This is formally computed by the following formula:

$$E' = \{ \langle u', v', l' \rangle \mid u', v' \in V' \wedge l' \in L' \wedge (u' \times v' \times l') \cap E \neq \emptyset \}$$

In illustration, Figure 2 shows a cyclic  $ORG$  for a subset of InterViews library, and Figure 4 the acyclic  $ORG'$  produced by this algorithm.

### 5.3 Breaking a Cycle

Given a cyclic  $ORG$ , such as the vertices and the associated edges in a cluster, how can one determine a test order to test the classes that are mutually dependent on each other?



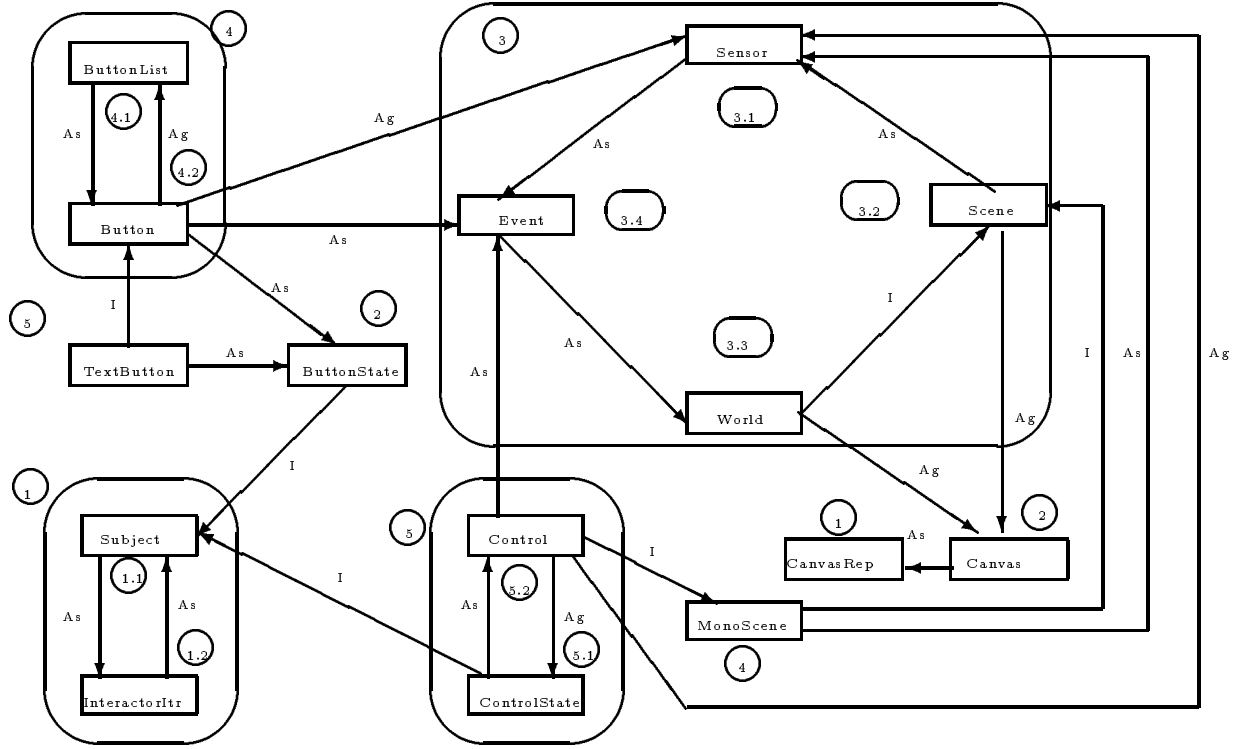


Figure 5: Test order for the subset of the InterViews library

edge and removing it until no cycle exists.

After breaking the cycles in each cluster, topological sorting can be applied to produce a test order. Figure 5 shows the major test order and minor test order for the InterViews example shown in Figure 4. The test order suggests that the classes be tested in the following order: Subject, InteractorItr, CanvasRep, ButtonState, Canvas, Sensor, Scene, World, Event, ButtonList, Button, MonoScene, TextButton, ControlState, Control. Since the generated test order is a topological order, it is possible that two or more vertices may have the same number. For example, the vertices Canvas and ButtonState in Figure 5 have the same major test order which is 2. In this case, either of them can be tested before the other.

## 5.4 Test Order Based Regression Test Strategy

### Test Order for Re-Testing Each Class

During regression testing, the previous test order algorithm can be used to find a cost-effective test order for the classes in a class firewall. Then, the generated test order is used as a guideline to select individual classes to conduct retests at the class unit level. Figure 6 shows the generated test order of the firewall for class Subject in Figure 3. The testing is carried out by following the major test order to test each cluster of classes. If the cluster to

be tested is a unit cluster, then its to be tested is a unit cluster<sup>4</sup>, then the member functions of the class in the cluster are tested according to their invocation dependencies. That is, if member function  $g$  invokes member function  $f$ , then  $f$  is tested before  $g$ . If recursive calls do not exist among the member functions, then topological sorting can be applied to produce a test order for testing the member functions. Otherwise, the termination condition for the recursive calls is identified and tested first, followed by testing of the other cases recursively. We do not address member function test order, interprocedural testing, and testing of recursive functions in this paper. The interested reader is referred to [Hwang 1988] [Harrold 1991] and other publications. If the cluster to be tested is a non-unit cluster<sup>5</sup>, then the classes in the cluster are tested according to the minor test order. Since the classes are cyclically dependent on each other, it is not possible to test all the member functions of each of the classes according to the minor test order in one pass. Therefore, the classes are tested in two or more passes. The trick is to test, in each pass, the member functions of a class that do not invoke other untested member functions. This process is repeated until all the member functions are tested. If recursion does not exist among the member functions of the classes, then two passes suffice; otherwise, recursive, multiple passes are needed.

The retest for a class may include: function member testing (both white-box or black-box), object state testing, and data-flow testing. The purpose is to check the different aspects of the target class, including its function members, objects states and behaviors, data flows.

### Test Strategy for Class Re-Integration

The generated test order for a class firewall can also be used as a test strategy for class re-integration. Class re-integration can be conducted based on the test order in the following steps:

1. build the initial system  $S$  by using the unaffected classes.
2. integrate  $S$  with the changed class  $C$  after it is tested.
3. follow the major test order to pick one class or one cluster to be re-integrated with  $S$ . If the chosen cluster has more than one classes, then they will be re-integrated following the minor test order.

The purpose of class re-integration testing is to check if the classes, which are related to each other or dependent on each other, can work together properly. The focus of this testing are: inherited members, aggregated parts, state dependent behaviors, and message passing between different objects. Similar to class unit testing, the generated test order indicates the re-integration order for the classes in the firewall.

---

<sup>4</sup>a unit cluster is a cluster which has only one class

<sup>5</sup>a non-unit cluster is a cluster which consists of more than one classes

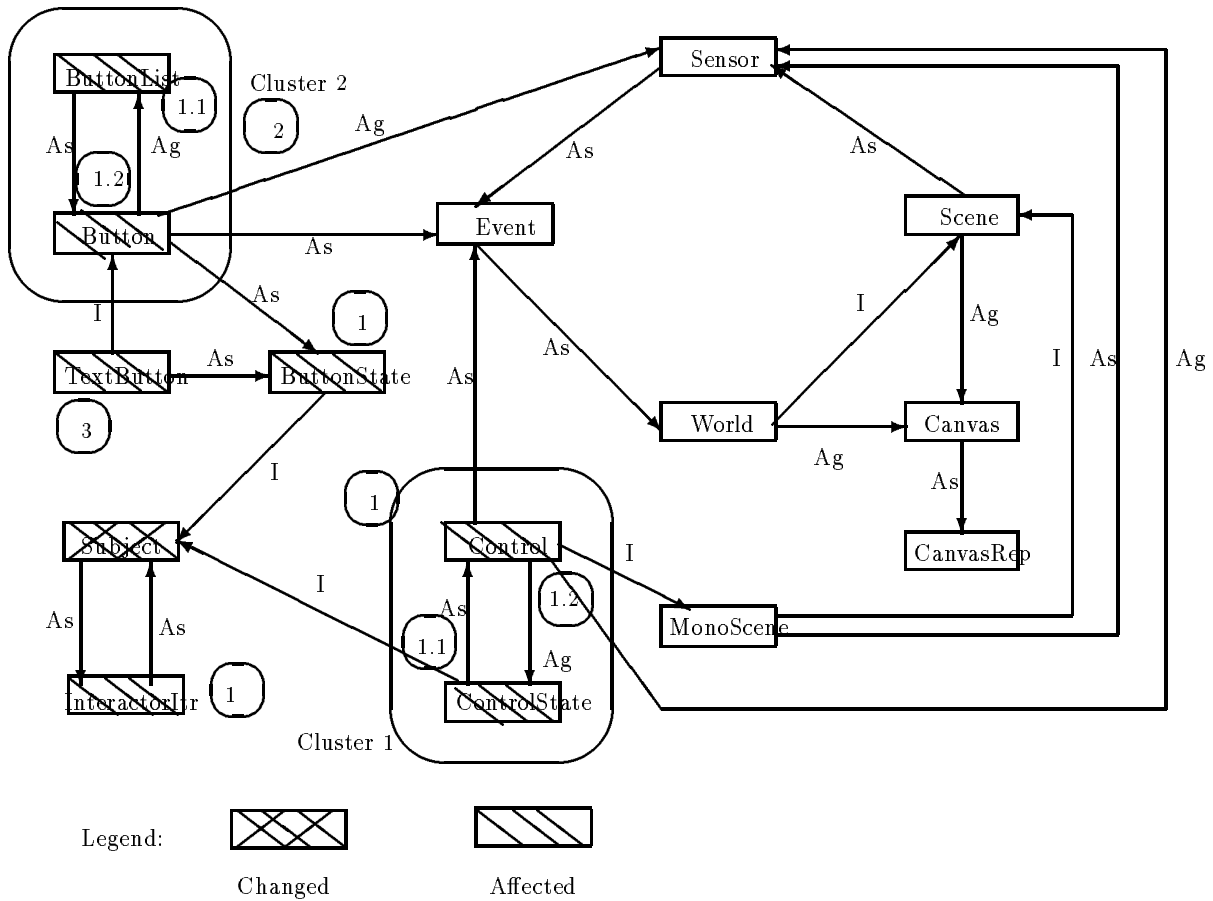


Figure 6: The Test Order for the Firewall of Class Subject

## 6 A Large Example

We have implemented an object-oriented testing environment using the reverse engineering approach [Kung 1993]. In the testing environment, a test support tool, called *firewall identifier*, is developed based on the class firewall concept to automatically identify all possibly affected classes for a changed class. After a class firewall is recognized, a *test order generator*, which is designed based on the results of this paper, is used to find the desired cost-effective test order for re-testing the affected classes. The generated test order provides not only a cost-effective class unit retest strategy, but also a detailed class re-integration strategy.

We have applied this tool to many applications, including the InterViews library. The InterViews library contains 147 files, more than 140 classes, and more than 400 relationships. Appendix A lists the classes in the InterViews library, and Appendix B shows the inheritance, aggregation and association relationships between these classes. Without the automatic support, it is very difficult for a regression tester to identify the relations between the classes.

How to conduct regression testing for this library when changes are made to one or more classes? For example, when a change is made in class Canvas, the regression tester will face the first challenge of identifying the affected classes. The tester only has two choice: either retest all classes, or manually find out statically or dynamically as the tests are performed. Both choices are costly and undesirable.

The *firewall identifier* program of the OOT environment provides one automatic solution to find all possibly affected classes for re-testing. For a changed class CanVas, the identified class firewall is shown in Figure 7.

After identifying the class firewall for a changed class, the tester starts to conduct the retests for the classes in the firewall. The first task is to retest each affected class at the class unit level. But, which class the tester should begin with ? Without a detailed guideline and automatic support tools, he will encounter another challenge: very costly test stubs. For example, if the Button class is chosen to be tested at first, then as shown in Figure 7, the test stubs, for classes: Sensor, ButtonState, Event and ButtonList, are needed. For constructing the test stub of ButtonState, the tester needs to understand some other classes, such as Subject, InteractorItr because the relationships between them. This fact suggests that a cost-effective strategy is needed for retest classes at the class unit level to save the test cost. The similar problem is also encountered by the tester when he performs the class re-integration tests. Thus, the real challenge problem the tester deals with are: 1) what is the cost-effective strategy for class unit retests ? 2) what is the desirable strategy for class re-integration ?

The *test order generator* in the OOT environment automatically generates a cost-effective test order for the classes in the firewall, so that the test cost on constructing test stubs for class unit regression testing and class re-integration testing is reduced. The lists below show the classes in the firewall for the Canvas class and the generated cost-effective test order for these classes. During class unit retests, the generated test order can be used as a strategy

to help the tester to find a cost-effective test sequence. Later, the test order can be used as a desirable step wise road map for class re-integration to reduce the test cost.

## Firewall for the InterViews Canvas class

class ID	class name	class ID	class name	class ID	class name	class ID	class name
1	Adjuster	47	GrowingVertices	88	RotatingLineList	114	SlidingLine
2	Banner	48	HBorder	89	RotatingRect	115	SlidingLineList
5	Border	49	HBox	90	RubberAxis	116	SlidingPointList
6	BorderFrame	50	HGlue	91	RubberCircle	117	SlidingRect
7	Box	51	HScroller	92	RubberClosedSpline	119	StretchingRect
10	BoxElement	52	Interactor	93	RubberEllipse	120	StringBrowser
13	Button	53	InteractorItr	94	RubberGroup	121	StringChooser
14	ButtonList	54	LeftMover	95	RubberHandles	122	StringEditor
16	Canvas	55	MarginFrame	96	RubberLine	126	Subject
18	Card	56	Menu	97	RubberPointList	130	TextButton
20	CheckBox	57	MenuBar	98	RubberRect	131	TextDisplay
25	Control	58	MenuItem	99	RubberSpline	132	TextEditor
26	ControlState	60	MonoScene	100	RubberSquare	133	TitleFrame
28	Deck	61	Mover	101	RubberVertex	134	TopLevel
31	Dialog	68	Painter	102	Rubberband	136	Tray
32	DownMover	69	Panner	103	ScalingLine	137	UpMover
33	Enlarger	71	Perspective	104	ScalingLineList	138	VBorder
34	Event	76	PulldownMenu	105	ScalingRect	139	VBox
38	FileChooser	77	PullrightMenu	106	Scene	140	VGlue
41	Frame	78	PushButton	107	Scroller	141	VScroller
42	Glue	79	RadioButton	108	Sensor	142	ViewList
43	GrowingBSpline	80	Raster	109	ShadowFrame	143	ViewPort
44	GrowingClosedBSpline	81	RasterRep	111	ShowFrame	144	World
45	GrowingMultiLine	82	Reducer	112	Slider	145	WorldView
46	GrowingPolygon	86	RightMover	113	SlidingEllipse	146	Zoomer
		87	RotatingLine				

## The Test Order of Classes in the Firewall for Canvas Class

class name	major order	minor order	class name	major order	minor order
Canvas	0	1	DownMover	4	0
RasterRep	1	2	Enlarger	4	0
Raster	1	3	GrowingBSpline	4	0
Painter	1	4	GrowingClosedBSpline	4	0
Event	1	5	GrowingMultiLine	4	0
Sensor	1	6	GrowingPolygon	4	0
Interactor	1	8	HBox	4	0
TopLevel	1	9	LeftMover	4	0
Scene	1	10	MarginFrame	4	0
World	1	11	Menu	4	0
ViewList	1	12	MenuItem	4	0
Perspective	1	13	Reducer	4	0
Adjuster	2	0	RightMover	4	0
Banner	2	0	RotatingLine	4	0
Border	2	0	RubberAxis	4	0
BoxElement	2	0	RubberCircle	4	0
Card	2	0	RubberSquare	4	0
Glue	2	0	RubberVertex	4	0
MonoScene	2	0	ScalingLine	4	0
Rubberband	2	0	ScalingLineList	4	0
Scroller	2	0	ScalingRect	4	0
Slider	2	0	ShadowFrame	4	0
TextDisplay	2	0	ShowFrame	4	0
Tray	2	0	SlidingEllipse	4	0
WorldView	2	0	SlidingLine	4	0
Subject	2	1	SlidingPointList	4	0
InteractorItr	2	2	SlidingRect	4	0
Box	3	0	StretchingRect	4	0
Deck	3	0	StringBrowser	4	0
Frame	3	0	StringEditor	4	0
GrowingVertices	3	0	UpMover	4	0
HBorder	3	0	VBox	4	0
HGlue	3	0	ButtonList	4	1
HScroller	3	0	Button	4	2
Mover	3	0	BorderFrame	5	0
Panner	3	0	FileBrowser	5	0
RotatingRect	3	0	MenuBar	5	0
RubberEllipse	3	0	PulldownMenu	5	0
RubberGroup	3	0	PullrightMenu	5	0
RubberLine	3	0	RotatingLineList	5	0
RubberPointList	3	0	RubberClosedSpline	5	0
RubberRect	3	0	RubberHandles	5	0

TextEditor	3	0	RubberSpline	5	0
VBorder	3	0	SlidingLineList	5	0
VGlue	3	0	StringChooser	5	0
VScroller	3	0	TextButton	5	0
Viewport	3	0	TitleFrame	5	0
Zoomer	3	0	CheckBox	6	0
ControlState	3	1	FileChooser	6	0
Control	3	2	PushButton	6	0
Dialog	4	0	RadioButton	6	0

## 7 The Regression Test Process

Regression testing can be carried out with respect to different levels of change and different types of change to an OO program. The levels of change include changes to member data, to member functions, to classes, and to subsystems or libraries. The types of change include changes to data values, to member function signatures, to conditional statements, and so on. It is beyond the scope of this paper to address the entire spectrum of changes and describe a total regression test process. Instead, we briefly describe a regression test process based on the results we have presented in this paper.

The regression test process consists of the following iterative phases:

Identification of Changed Classes. In this phase, the tester compares the original program and the modified program to identify classes that are changed. This can be easily achieved by using a code comparator.

Identification of Affected Classes. In this phase, the tester identifies affected classes. This is achieved by computing the class firewall(s) for the classes that are changed.

Generation of a Class Test Order. In this phase, the tester generates a class test order to facilitate test case preparation and the actual conduct of regression testing. By using the algorithm and tool presented in this paper, this task can be easily accomplished.

Selection of Test Cases. In this phase, the tester selects test cases to retest the affected classes. This is done according to the test order and by consulting the test plan if one exists. The test order specifies which classes should be tested before which other classes so that effort needed to construct test drivers and test stubs can be saved. It also facilitates test case reuse, meaning that some test cases of a class may be reused by its dependent class. For example, in certain circumstances, a test case for a member function of a superclass, say the *rotate* function of *polygon*, can be reused to test the same function inherited by a subclass, say *rectangle*. A test plan contains test cases that were previously designed to test the original program. Test cases that were designed to test the affected classes are selected.

Test Case Modification and Generation. In this phase, the tester modifies the selected test cases and generates new test cases according to the changes made. For example, if changes have been made to change the structure of the original program without affecting its functionality, then the relevant structural test cases are modified so that the required coverage criteria can be met. If the functionality of the original program has been enhanced, then

the relevant functional test cases are modified and perhaps new test cases are generated to test the new functionality. The result of this phase is a new test plan, which contains the modified test cases and the new test cases.

*Regression Test Execution.* In this phase, the tester prepares test data according to the test cases and execute the modified program using the test data. The test results then are analyzed and documented. The modified program is tested at different levels: unit class testing, class integration testing, and system testing. In unit class testing, the classes in the firewall(s) are tested according to the test order, while in class integration testing, the classes are integrated according to the test order. Again, since the test order is generated according to the dependencies between the classes, using the test order to guide unit class testing and class integration testing will reduce the effort needed to construct the test drivers and test stubs.

## 8 Conclusions and Future Work

We have discussed the regression processes and addressed problems in regression testing of object-oriented programs. We have presented methods for finding the affected classes when changes are made to some classes of a C++ program. We also presented a test order generation algorithm to provide a cost-effective test sequence of components to serve as a road map for a tester to retest classes in the class firewall. The results can be used to conduct cost-effective regression testing of classes at the class unit level and the class integration level. These techniques have been incorporated in our object-oriented testing environment and have been applied to realistic applications, like the InterViews class library. The results so far are promising. We are currently investigating guidelines for find a firewall of a changed class member, and algorithms for testing the member functions of the classes.

## 9 Acknowledgment

This research is supported by Fujitsu Network Transmission Systems, Inc. and Hewlett-Packard Company. The authors want to thank the other members of the Object-Oriented Testing Team, K. Chang, M. Cheng, S. Ha, K. Huang, J. Samuel, Y. Song, and N. Suchak, for many stimulating discussions and their effort to implement the environment.

## 10 References

[Aho 1983] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms," Addison-Wesley Publ. Comp., 1983.

- [Beizer 1990] B. Beizer, "Software Testing Techniques," 2nd ed., Van Nostrand Reinhold, 1990.
- [Benedusi 1988] P. Benedusi, A. Cimitile, and U. De carlini, "Post-maintenance testing based on path change analysis," Proc. IEEE Conf. on Software Maintenance, pp. 352 – 361, 1988.
- [Booch 1991] G. Booch, "Object-Oriented Design with Applications" Redwood City, Calif.: Benjamin/Cummings, 1991.
- [Coad 1990] P. Coad and Yourdon, E. "Object-oriented Analysis." Yourdon Press, 1990.
- [Fiedler 1989] S. P. Fiedler, "Object-oriented unit testing," Hewlett-Packard Journal, pp. 69 - 74, April 1989.
- [Fischer 1977] K.F. Fischer, "A Test Case Selection Method for the Validation of Software Maintenance Modifications", IEEE COMPSAC 77 Int. Conf. Procs., pp. 421-426, No. 1977.
- [Fischer 1981] K.F. Fischer, F. Raji and A. Chruscicki, "A Methodology for Re-Testing Modified Software", National Telecomms. Conf. Procs., pp. B6.3.1-6, Nov. 1981.
- [Gao 1993] Jerry Gao, "Test Order for Object-Oriented Programs," Technical Report No. 93-1, Software Engineering Center for Telecommunications, The University of Texas at Arlington, 1993.
- [Harrold 1988] M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance", Proc. Conf. Software Maintenance, pp. 362-367, Phoenix, 1988.
- [Harrold 1989] M. J. Harrold and M. L. Soffa, "Interprocedural data flow testing", Proc. Conf. Software Maintenance, pp. 362-367, Phoenix, 1988.
- [Harrold 1991] M. J. Harrold and J. D. McGregor, "Toward a testing methodology for object-oriented software systems," Department of Computer Science, Clemson University.
- [Harrold 1992] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object-oriented class structure", Proc. of 14th International Conf. on Software Engineering, 1992.
- [Hartmann 1988] J. Hartmann and D. J. Robson, "Approaches to Regression Testing", Proc. Conf. Software Maintenance, pp. 368-372, 1988.
- [Hwang 1988] J. C. Hwang, M. W. Du, and C. R. Chou, "Finding program slices for recursive procedures," Proc. of 9th International Conf. on Software Engineering, pp. 220 – 227, 1988.
- [Laski 1992] Janusz Laski and Wojciech Szermer, "Identification of Program Modifications and its Applications in Software Maintenance", Proc. Conf. Software Maintenance, pp. 282-290, 1992.
- [Korson 1990] T. Korson and J. D. McGregor. "Understanding object- oriented: a unifying paradigm." CACM Vol. 33, No. 9, pp. 40 - 60, Sept. 1990.

- [Kung 1993] D. Kung, "The behavior network model for conceptual information modeling." *Journal of Information Systems*, Vol. 18, No. 1, pp. 1 – 21, 1993.
- [Lee 1990] J. Lee and X. He, "A methodology for test selection," *Journal of Systems and Software*, Vol. 13, pp. 177 - 185, 1990.
- [Leung 1989] H.K.N Leung and L. White, "Insights into regression testing", *Proc. Conf. Software Maintenance*, pp. 60-69, Miami, FL, Oct. 1989.
- [Leung 1990] H. K. N. Leung and L. White, "A study of integration testing and software regression at the integration level," *Proc. IEEE Conf. on Software maintenance*, pp. 290 – 301, 1990.
- [Lippman 1991] S. B. Lippman, "A C++ Primer," Reading, Mass: Addison-Wesley, 1991.
- [Perry 1990] D. E. Perry and G. E. Kaiser, "Adequate testing and object-oriented programming," *Journal of Object-Oriented Programming*, Vol. 2, pp. 13 - 19, January/February 1990.
- [Prather 1987] Ronald E. Prather and J. Paul Myers, JR, "The Path Prefix Software Testing Strategy", *IEEE Transactions On Software Engineering*, Vol. SE-13, NO. 7, July 1987.
- [Rumbaugh 1991] J. Rumbaugh et al., "Object-Oriented Modeling and Design," Prentice-Hall, 1991.
- [Smith 1990] M. D. Smith and D. J. Robson, "Object-oriented programming — the problems of validation," *Proc. IEEE Conference on Software Maintenance — 1990*. pp. 272 – 281.
- [Stroustrup 1991] B. Stroustrup, "The C++ Programming Language", 2nd ed., Addison-Wesley, 1991.
- [Weyuker 1986] E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Trans. on Software Eng.*, Vol. SE-12, No. 12, pp. 1128 - 1138, 1986.
- [White 1992] L. White and H.K.N. Leung, "A Firewall Concept for both Control-Flow and Data-Flow in Regression Integration Testing", *Proc. Conf. Software Maintenance*, pp. 262-271, 1992.
- [Wilde 1991] N. Wilde and R. Huitt, "Issues in the maintenance of object-oriented programs," University of West Florida and Bell Communications Research, 1991.
- [Wilde 1992] N. Wilde and R. Huitt, "Maintenance support for object-oriented programs," *IEEE Trans. on Software Eng.*, Vol. 18, No. 12, pp. 1038 - 1044, Dec. 1992.

# Appendix A: Class List for the InterViews Library

Number of Files = 147; Number of Classes = 146; Number of Relationships > 400.

class ID	class name	class ID	class name	class ID	class name
1	Adjuster	49	HBox	97	RubberPointList
2	Banner	50	HGlue	98	RubberRect
3	Bitmap	51	HScroller	99	RubberSpline
4	BitmapRep	52	Interactor	100	RubberSquare
5	Border	53	InteractorItr	101	RubberVertex
6	BorderFrame	54	LeftMover	102	Rubberband
7	Box	55	MarginFrame	103	ScalingLine
8	BoxCanonical	56	Menu	104	ScalingLineList
9	BoxDimension	57	MenuBar	105	ScalingRect
10	BoxElement	58	MenuItem	106	Scene
11	Brush	59	Message	107	Scroller
12	BrushRep	60	MonoScene	108	Sensor
13	Button	61	Mover	109	ShadowFrame
14	ButtonList	62	ObjectSpace	110	Shape
15	ButtonState	63	ObjectStub	111	ShowFrame
16	Canvas	64	ObjectTable	112	Slider
17	CanvasRep	65	ObjectTableEntry	113	SlidingEllipse
18	Card	66	OptionDesc	114	SlidingLine
19	Catalog	67	Packet	115	SlidingLineList
20	CheckBox	68	Painter	116	SlidingPointList
21	ChiefDeputy	69	Panner	117	SlidingRect
22	Color	70	Pattern	118	SpaceManager
23	ColorRep	71	Perspective	119	StretchingRect
24	Connection	72	PopupMenu	120	StringBrowser
25	Control	73	PropertyData	121	StringChooser
26	ControlState	74	PropertyDef	122	StringEditor
27	Cursor	75	PropertySheet	123	StringId
28	Deck	76	PulldownMenu	124	StringPool
29	DefaultProps	77	PullrightMenu	125	StringTable
30	Deputy	78	PushButton	126	Subject
31	Dialog	79	RadioButton	127	TGlue
32	DownMover	80	Raster	128	Table
33	Enlarger	81	RasterRep	129	TextBuffer
34	Event	82	Reducer	130	TextButton
35	EventList	83	Regexp	131	TextDisplay
36	FBDirectory	84	ReqErr	132	TextEditor
37	FileBrowser	85	Resource	133	TitleFrame
38	FileChooser	86	RightMover	134	TopLevel
39	Font	87	RotatingLine	135	Transformer
40	FontRep	88	RotatingLineList	136	Tray
41	Frame	89	RotatingRect	137	UpMover
42	Glue	90	RubberAxis	138	VBorder
43	GrowingBSpline	91	RubberCircle	139	VBox
44	GrowingClosedBSpline	92	RubberClosedSpline	140	VGlue
45	GrowingMultiLine	93	RubberEllipse	141	VScroller
46	GrowingPolygon	94	RubberGroup	142	ViewList
47	GrowingVertices	95	RubberHandles	143	Viewport
48	HBorder	96	RubberLine	144	World
				145	WorldView
				146	Zoomer

# Appendix B: Class Relations in the InterViews Library

The Inheritance Relation between Classes in InterViews Library

(class ID, class ID, R)	(class ID, class ID, R)	(class ID, class ID, R)	(class ID, class ID, R)
(1, 52, R <sub>I</sub> )	(146, 1, R <sub>I</sub> )	(33, 146, R <sub>I</sub> )	(82, 146, R <sub>I</sub> )
(61, 1, R <sub>I</sub> )	(54, 61, R <sub>I</sub> )	(86, 61, R <sub>I</sub> )	(137, 61, R <sub>I</sub> )
(32, 61, R <sub>I</sub> )	(2, 52, R <sub>I</sub> )	(3, 85, R <sub>I</sub> )	(5, 52, R <sub>I</sub> )
(48, 5, R <sub>I</sub> )	(138, 5, R <sub>I</sub> )	(7, 106, R <sub>I</sub> )	(49, 7, R <sub>I</sub> )
(139, 7, R <sub>I</sub> )	(11, 85, R <sub>I</sub> )	(15, 126, R <sub>I</sub> )	(13, 52, R <sub>I</sub> )
(130, 13, R <sub>I</sub> )	(78, 130, R <sub>I</sub> )	(79, 130, R <sub>I</sub> )	(20, 130, R <sub>I</sub> )
(22, 85, R <sub>I</sub> )	(25, 60, R <sub>I</sub> )	(26, 126, R <sub>I</sub> )	(28, 106, R <sub>I</sub> )
(31, 60, R <sub>I</sub> )	(37, 120, R <sub>I</sub> )	(38, 121, R <sub>I</sub> )	(39, 85, R <sub>I</sub> )
(40, 85, R <sub>I</sub> )	(41, 60, R <sub>I</sub> )	(111, 41, R <sub>I</sub> )	(133, 111, R <sub>I</sub> )
(541, 111, R <sub>I</sub> )	(109, 41, R <sub>I</sub> )	(55, 41, R <sub>I</sub> )	(42, 52, R <sub>I</sub> )
(50, 42, R <sub>I</sub> )	(140, 42, R <sub>I</sub> )	(58, 25, R <sub>I</sub> )	(56, 25, R <sub>I</sub> )
(57, 49, R <sub>I</sub> )	(76, 56, R <sub>I</sub> )	(77, 56, R <sub>I</sub> )	(72, 56, R <sub>I</sub> )
(59, 52, R <sub>I</sub> )	(68, 85, R <sub>I</sub> )	(69, 60, R <sub>I</sub> )	(112, 52, R <sub>I</sub> )
(70, 85, R <sub>I</sub> )	(71, 85, R <sub>I</sub> )	(80, 85, R <sub>I</sub> )	(102, 85, R <sub>I</sub> )
(93, 102, R <sub>I</sub> )	(113, 93, R <sub>I</sub> )	(91, 93, R <sub>I</sub> )	(97, 102, R <sub>I</sub> )
(101, 97, R <sub>I</sub> )	(95, 101, R <sub>I</sub> )	(99, 101, R <sub>I</sub> )	(92, 101, R <sub>I</sub> )

(116, 97, $R_I$ )	(115, 116, $R_I$ )	(104, 97, $R_I$ )	(88, 104, $R_I$ )
(94, 102, $R_I$ )	(96, 102, $R_I$ )	(90, 96, $R_I$ )	(114, 96, $R_I$ )
(103, 96, $R_I$ )	(87, 96, $R_I$ )	(98, 102, $R_I$ )	(100, 98, $R_I$ )
(117, 98, $R_I$ )	(119, 98, $R_I$ )	(105, 98, $R_I$ )	(89, 102, $R_I$ )
(47, 102, $R_I$ )	(45, 47, $R_I$ )	(46, 47, $R_I$ )	(43, 47, $R_I$ )
(44, 47, $R_I$ )	(106, 52, $R_I$ )	(60, 106, $R_I$ )	(107, 52, $R_I$ )
(51, 107, $R_I$ )	(141, 107, $R_I$ )	(108, 85, $R_I$ )	(62, 63, $R_I$ )
(118, 30, $R_I$ )	(120, 52, $R_I$ )	(121, 31, $R_I$ )	(122, 52, $R_I$ )
(63, 85, $R_I$ )	(126, 85, $R_I$ )	(132, 52, $R_I$ )	(135, 85, $R_I$ )
(136, 106, $R_I$ )	(143, 60, $R_I$ )	(144, 106, $R_I$ )	(145, 52, $R_I$ )

### The Aggregation Relation between Classes in InterViews Library

(class ID, class ID, $R$ )	(class ID, class ID, $R$ )	(class ID, class ID, $R$ )	(class ID, class ID, $R$ )
(68, 3, $R_{AG}$ )	(106, 16, $R_{AG}$ )	(144, 16, $R_{AG}$ )	(144, 22, $R_{AG}$ )
(144, 39, $R_{AG}$ )	(144, 68, $R_{AG}$ )	(145, 16, $R_{AG}$ )	(1, 71, $R_{AG}$ )
(1, 108, $R_{AG}$ )	(1, 68, $R_{AG}$ )	(2, 68, $R_{AG}$ )	(3, 4, $R_{AG}$ )
(3, 135, $R_{AG}$ )	(7, 10, $R_{AG}$ )	(7, 52, $R_{AG}$ )	(13, 108, $R_{AG}$ )
(13, 14, $R_{AG}$ )	(130, 68, $R_{AG}$ )	(79, 3, $R_{AG}$ )	(25, 26, $R_{AG}$ )
(25, 108, $R_{AG}$ )	(28, 18, $R_{AG}$ )	(28, 71, $R_{AG}$ )	(28, 52, $R_{AG}$ )
(37, 36, $R_{AG}$ )	(1, 52, $R_{AG}$ )	(8, 9, $R_{AG}$ )	(29, 74, $R_{AG}$ )
(35, 34, $R_{AG}$ )			

### The Association Relation between Classes in InterViews Library

(class ID, class ID, $R$ )	(class ID, class ID, $R$ )	(class ID, class ID, $R$ )	(class ID, class ID, $R$ )
(127, 110, $R_{AS}$ )	(121, 120, $R_{AS}$ )	(136, 52, $R_{AS}$ )	(121, 52, $R_{AS}$ )
(143, 68, $R_{AS}$ )	(122, 129, $R_{AS}$ )	(145, 144, $R_{AS}$ )	(122, 15, $R_{AS}$ )
(10, 52, $R_{AS}$ )	(122, 131, $R_{AS}$ )	(14, 13, $R_{AS}$ )	(125, 123, $R_{AS}$ )
(18, 52, $R_{AS}$ )	(142, 52, $R_{AS}$ )	(134, 27, $R_{AS}$ )	(126, 142, $R_{AS}$ )
(134, 52, $R_{AS}$ )	(53, 52, $R_{AS}$ )	(134, 3, $R_{AS}$ )	(53, 142, $R_{AS}$ )
(1, 3, $R_{AS}$ )	(65, 24, $R_{AS}$ )	(11, 12, $R_{AS}$ )	(65, 63, $R_{AS}$ )
(13, 15, $R_{AS}$ )	(64, 63, $R_{AS}$ )	(16, 17, $R_{AS}$ )	(64, 65, $R_{AS}$ )
(19, 63, $R_{AS}$ )	(131, 68, $R_{AS}$ )	(21, 24, $R_{AS}$ )	(131, 16, $R_{AS}$ )
(22, 23, $R_{AS}$ )	(132, 129, $R_{AS}$ )	(26, 25, $R_{AS}$ )	(132, 131, $R_{AS}$ )
(27, 22, $R_{AS}$ )	(1, 68, $R_{AS}$ )	(30, 24, $R_{AS}$ )	(1, 34, $R_{AS}$ )
(30, 21, $R_{AS}$ )	(1, 110, $R_{AS}$ )	(31, 15, $R_{AS}$ )	(146, 52, $R_{AS}$ )
(34, 52, $R_{AS}$ )	(146, 68, $R_{AS}$ )	(34, 144, $R_{AS}$ )	(146, 34, $R_{AS}$ )
(38, 52, $R_{AS}$ )	(33, 52, $R_{AS}$ )	(38, 37, $R_{AS}$ )	(33, 68, $R_{AS}$ )
(38, 55, $R_{AS}$ )	(82, 52, $R_{AS}$ )	(39, 40, $R_{AS}$ )	(82, 68, $R_{AS}$ )
(133, 2, $R_{AS}$ )	(61, 52, $R_{AS}$ )	(133, 52, $R_{AS}$ )	(61, 68, $R_{AS}$ )
(52, 110, $R_{AS}$ )	(61, 34, $R_{AS}$ )	(52, 27, $R_{AS}$ )	(54, 52, $R_{AS}$ )
(52, 106, $R_{AS}$ )	(54, 68, $R_{AS}$ )	(52, 144, $R_{AS}$ )	(86, 52, $R_{AS}$ )
(52, 16, $R_{AS}$ )	(86, 68, $R_{AS}$ )	(52, 71, $R_{AS}$ )	(137, 52, $R_{AS}$ )
(52, 3, $R_{AS}$ )	(137, 68, $R_{AS}$ )	(52, 108, $R_{AS}$ )	(32, 52, $R_{AS}$ )
(52, 68, $R_{AS}$ )	(32, 68, $R_{AS}$ )	(52, 123, $R_{AS}$ )	(2, 68, $R_{AS}$ )
(52, 134, $R_{AS}$ )	(3, 39, $R_{AS}$ )	(56, 52, $R_{AS}$ )	(3, 135, $R_{AS}$ )
(56, 26, $R_{AS}$ )	(68, 3, $R_{AS}$ )	(56, 106, $R_{AS}$ )	(3, 4, $R_{AS}$ )
(56, 144, $R_{AS}$ )	(68, 4, $R_{AS}$ )	(57, 26, $R_{AS}$ )	(4, 39, $R_{AS}$ )
(68, 22, $R_{AS}$ )	(4, 135, $R_{AS}$ )	(68, 70, $R_{AS}$ )	(5, 68, $R_{AS}$ )
(68, 11, $R_{AS}$ )	(48, 68, $R_{AS}$ )	(68, 39, $R_{AS}$ )	(138, 68, $R_{AS}$ )
(68, 135, $R_{AS}$ )	(7, 52, $R_{AS}$ )	(69, 52, $R_{AS}$ )	(7, 110, $R_{AS}$ )
(112, 52, $R_{AS}$ )	(7, 8, $R_{AS}$ )	(112, 71, $R_{AS}$ )	(49, 52, $R_{AS}$ )
(71, 142, $R_{AS}$ )	(49, 110, $R_{AS}$ )	(80, 22, $R_{AS}$ )	(49, 8, $R_{AS}$ )
(80, 81, $R_{AS}$ )	(139, 52, $R_{AS}$ )	(102, 68, $R_{AS}$ )	(139, 110, $R_{AS}$ )
(102, 16, $R_{AS}$ )	(139, 8, $R_{AS}$ )	(94, 102, $R_{AS}$ )	(68, 12, $R_{AS}$ )
(106, 52, $R_{AS}$ )	(13, 34, $R_{AS}$ )	(60, 52, $R_{AS}$ )	(13, 68, $R_{AS}$ )
(107, 52, $R_{AS}$ )	(130, 15, $R_{AS}$ )	(107, 71, $R_{AS}$ )	(130, 68, $R_{AS}$ )
(107, 108, $R_{AS}$ )	(78, 15, $R_{AS}$ )	(62, 24, $R_{AS}$ )	(78, 68, $R_{AS}$ )
(62, 63, $R_{AS}$ )	(79, 15, $R_{AS}$ )	(62, 19, $R_{AS}$ )	(79, 68, $R_{AS}$ )
(62, 64, $R_{AS}$ )	(20, 15, $R_{AS}$ )	(62, 118, $R_{AS}$ )	(20, 68, $R_{AS}$ )
(118, 24, $R_{AS}$ )	(16, 22, $R_{AS}$ )	(120, 15, $R_{AS}$ )	(106, 16, $R_{AS}$ )
(120, 131, $R_{AS}$ )	(144, 16, $R_{AS}$ )	(121, 122, $R_{AS}$ )	(145, 16, $R_{AS}$ )
(68, 16, $R_{AS}$ )	(96, 68, $R_{AS}$ )	(25, 52, $R_{AS}$ )	(90, 16, $R_{AS}$ )
(25, 26, $R_{AS}$ )	(90, 68, $R_{AS}$ )	(25, 34, $R_{AS}$ )	(114, 16, $R_{AS}$ )
(27, 3, $R_{AS}$ )	(114, 68, $R_{AS}$ )	(27, 39, $R_{AS}$ )	(103, 16, $R_{AS}$ )
(28, 68, $R_{AS}$ )	(103, 68, $R_{AS}$ )	(28, 71, $R_{AS}$ )	(87, 16, $R_{AS}$ )
(28, 52, $R_{AS}$ )	(87, 68, $R_{AS}$ )	(31, 52, $R_{AS}$ )	(98, 16, $R_{AS}$ )
(31, 34, $R_{AS}$ )	(98, 68, $R_{AS}$ )	(108, 34, $R_{AS}$ )	(100, 16, $R_{AS}$ )
(52, 34, $R_{AS}$ )	(100, 68, $R_{AS}$ )	(37, 15, $R_{AS}$ )	(117, 16, $R_{AS}$ )
(38, 15, $R_{AS}$ )	(117, 68, $R_{AS}$ )	(68, 40, $R_{AS}$ )	(119, 16, $R_{AS}$ )
(41, 52, $R_{AS}$ )	(119, 68, $R_{AS}$ )	(111, 52, $R_{AS}$ )	(105, 16, $R_{AS}$ )
(111, 34, $R_{AS}$ )	(105, 68, $R_{AS}$ )	(541, 52, $R_{AS}$ )	(89, 16, $R_{AS}$ )
(109, 52, $R_{AS}$ )	(89, 68, $R_{AS}$ )	(55, 52, $R_{AS}$ )	(47, 16, $R_{AS}$ )
(42, 68, $R_{AS}$ )	(47, 68, $R_{AS}$ )	(50, 68, $R_{AS}$ )	(45, 16, $R_{AS}$ )
(140, 68, $R_{AS}$ )	(45, 68, $R_{AS}$ )	(16, 52, $R_{AS}$ )	(46, 16, $R_{AS}$ )
(144, 52, $R_{AS}$ )	(46, 68, $R_{AS}$ )	(58, 52, $R_{AS}$ )	(43, 16, $R_{AS}$ )

(56,	25,	RAS)	(43,	68,	RAS)	(56,	34,	RAS)	(44,	16,	RAS)
(57,	25,	RAS)	(44,	68,	RAS)	(76,	52,	RAS)	(106,	68,	RAS)
(77,	52,	RAS)	(106,	108,	RAS)	(68,	80,	RAS)	(60,	68,	RAS)
(69,	68,	RAS)	(60,	108,	RAS)	(112,	68,	RAS)	(107,	68,	RAS)
(112,	34,	RAS)	(51,	52,	RAS)	(112,	110,	RAS)	(51,	68,	RAS)
(70,	3,	RAS)	(51,	108,	RAS)	(71,	52,	RAS)	(51,	34,	RAS)
(75,	74,	RAS)	(51,	71,	RAS)	(80,	16,	RAS)	(141,	52,	RAS)
(68,	81,	RAS)	(141,	68,	RAS)	(81,	16,	RAS)	(141,	108,	RAS)
(93,	16,	RAS)	(141,	34,	RAS)	(93,	68,	RAS)	(141,	71,	RAS)
(113,	16,	RAS)	(120,	34,	RAS)	(113,	68,	RAS)	(120,	71,	RAS)
(91,	16,	RAS)	(121,	15,	RAS)	(91,	68,	RAS)	(121,	34,	RAS)
(97,	16,	RAS)	(122,	34,	RAS)	(97,	68,	RAS)	(63,	24,	RAS)
(101,	16,	RAS)	(126,	52,	RAS)	(101,	68,	RAS)	(126,	53,	RAS)
(95,	16,	RAS)	(53,	126,	RAS)	(95,	68,	RAS)	(64,	24,	RAS)
(99,	16,	RAS)	(129,	83,	RAS)	(99,	68,	RAS)	(132,	34,	RAS)
(92,	16,	RAS)	(132,	71,	RAS)	(92,	68,	RAS)	(136,	68,	RAS)
(116,	16,	RAS)	(136,	108,	RAS)	(116,	68,	RAS)	(136,	110,	RAS)
(115,	16,	RAS)	(136,	127,	RAS)	(115,	68,	RAS)	(143,	52,	RAS)
(104,	16,	RAS)	(143,	108,	RAS)	(104,	68,	RAS)	(143,	71,	RAS)
(88,	16,	RAS)	(144,	66,	RAS)	(88,	68,	RAS)	(144,	73,	RAS)
(94,	16,	RAS)	(145,	27,	RAS)	(94,	68,	RAS)	(145,	110,	RAS)
(96,	16,	RAS)	(52,	35,	RAS)						

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related Work</b>	<b>3</b>
<b>3</b>	<b>The Class Regression Test Model</b>	<b>4</b>
3.1	Class Relations . . . . .	5
3.2	Notion of Object Relation Graph . . . . .	7
<b>4</b>	<b>Class Firewall</b>	<b>9</b>
4.1	The Notion of a Class Firewall . . . . .	10
4.2	Constructing a Class Firewall . . . . .	10
<b>5</b>	<b>Test Order for Class Firewalls</b>	<b>12</b>
5.1	Overview of the test Order Finding Algorithm . . . . .	15
5.2	Converting a Cyclic ORG to an Acyclic ORG Using Clusters . . . . .	16
5.3	Breaking a Cycle . . . . .	17
5.4	Test Order Based Regression Test Strategy . . . . .	19
<b>6</b>	<b>A Large Example</b>	<b>22</b>
<b>7</b>	<b>The Regression Test Process</b>	<b>24</b>
<b>8</b>	<b>Conclusions and Future Work</b>	<b>25</b>
<b>9</b>	<b>Acknowledgment</b>	<b>25</b>
<b>10</b>	<b>References</b>	<b>25</b>