A Test Strategy for Object-Oriented Programs

David Kung, Jerry Gao, Pei Hsia
Computer Science Engineering Dept.
The Univ. of Texas at Arlington
P. O. Box 19015
Arlington, TX 76019-0015
Yasufumi Toyoshima, Cris Chen
Fujitsu Network Transmission Systems, Inc.

March 25, 1997

Abstract

The complexity and interdependencies of an object-oriented program makes testing of such programs difficult. In this paper, we present a reverse engineering based model called Object Relation Diagram (ORD), which is generated from analyzing the C++ source code of an object-oriented program. An ORD is a directed graph in which the vertices represent the object classes and the edges represent the relationships among the object classes. Based on the ORD, a test strategy, called test order, for unit testing and integration testing of object-oriented programs is described. The test order algorithm uses topological sorting and clusters of strongly connected subgraphs of the ORD. It computes the optimal test order in the sense that the effort required to construct the test stubs to simulate the untested classes/member functions is minimum. We show the savings of the test strategy through statistics of the InterViews library.
1 Introduction

The object-oriented (OO) paradigm is rapidly expanding its acceptance in the software industry. However, the powerful features of this new paradigm also introduce a new set of testing and maintenance problems. The inheritance, aggregation, and association relationships among the object classes make an OO program difficult to understand and test. The encapsulation and information hiding features result in chains of member function invocations that often involve objects of more than one class\(^1\). Potential problems for software testing are: 1) it is difficult to understand the code and prepare test cases; 2) it is not cost-effective to construct test stubs for member functions since most of them consist of one to two statements [wild91a]. Rather, one would just use them provided that they have been (adequately) tested\(^2\); 3) it is necessary to determine and limit the required regression tests when a function or a class is changed; 4) it requires a fresh look into the traditional coverage criteria and to extend them to include not just coverage of individual functions, but also function invocation sequences, object states and state sequences, and data definition and use paths across functions and objects.

In this paper, we first discuss two of the problems in OO testing in greater detail in section 2. We then present a tool called Object Relation Diagram (ORD) in section 3. The ORD is a reverse engineering tool that extracts from C++ source code the various relationships among the classes and displays these relationships in Rumbaugh et al.'s notation. Section 4 presents a test order generation algorithm that generates an optimal class test order for unit testing and integration testing of the classes of an OO program. The test order algorithm computes the optimal test order based on the dependencies among the classes. We then present our experience with the tool and

\(^1\)This paper is written with C++ in mind. However, the results are not limited to C++.

\(^2\)Stubs may be needed for chains of function invocations if the chains are frequently executed and the execution time is considerably long.
the test order generator in section 5. We show that the tools can provide substantial savings in unit testing and integration testing. Section 6 presents the conclusions.

2 The Problems

The understanding problem is introduced by the encapsulation and information hiding features. Encapsulation means modeling and storing, with an object, the attributes and operations an object is capable of performing [kors90a]. Information hiding [parn72a] means only the operations of an object can access to its private part. These features result in the so-called “delocalized plan” [solo88a], in which several member functions from possibly several object classes are invoked to achieve an intended functionality. Often, a member function of a class may in turn invoke other member functions, resulting in the so-called invocation chain of member functions [wild92a].

Table 1 summarizes the number of invocation chains of different lengths for the InterViews library. The result does not take into consideration overloading, polymorphism, and dynamic binding3 because these features require actual execution of the member functions. We see that there are 4,818 member function invocation chains. The longest chain involves 14 member function calls in sequence and the majority of cases involve chains of two to nine member functions.

Table 1: InterViews invocation chains

<table>
<thead>
<tr>
<th>Invocation chain length</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of chains</td>
<td>246</td>
<td>305</td>
<td>568</td>
<td>783</td>
<td>742</td>
<td>492</td>
<td>514</td>
<td>482</td>
<td>330</td>
<td>172</td>
<td>82</td>
<td>59</td>
<td>31</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

3When these features are considered, then the invocation chains will be longer and more difficult to comprehend because one does not know which code segment is actually executed at run time.
The implication of the invocation chains is that a tester has to understand sequences of member functions and the semantics of the classes prior to preparing any test cases to test a member function. Since it is necessary to understand all the parts in sufficient detail before testing, this adds tremendous complexity to testing of OO systems.

We have also conducted an experiment to find out how much time a tester would spend to test three simple, small member functions from an elevator program written in C++. The testers are students who have learned and programmed in C++. They are required to manually prepare basis path [pres92a] test cases and test data for testing the member functions. Although each of the member functions are very small (totally 15 lines of code or in average 5 lines per member function) it took in average 0.95, 0.93, and 0.79 person-hour to prepare test data, test driver, and test stub, respectively. We know that automatic test data and test driver generation is possible and this would save about 1.88 (i.e., 0.95 + 0.93) hours per member function. However, automatic test stub generation is not possible because this requires understanding of the semantics of the called functions. Therefore, reducing the effort needed to construct test stubs will result in additional savings.

The dependency problem is caused by the complex relationships that exist in an OO program. These include the inheritance, aggregation, association, template class instantiation, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding relationships. These relationships imply that one class inevitably depends on another class. For example, the inheritance relationship implies that a derived class reuses both the data members and function members of a base class; and hence, it is dependent on the base class. Another type of dependency is the member function invocation dependency, as illustrated above.

We have investigated a version of the InterViews library, which contains 122 classes, more than 400
inheritance, aggregation, and association relationships, and more than 1,000 member functions. It
does not contain template class or nested class. However, the 122 classes are related to each other,
forming a connected cyclic graph. Another industrial library, i.e., the IBM OS2 Class Library, we
have examined contains many template classes, some of which are derived classes of other template
classes. The InterViews library is considered a small library, compared to some of the class libraries
nowadays exist in the industry.

The complex relationships that exist in an OO program make testing and maintenance extremely
difficult. Below are some observations: 1) It is extremely difficult to understand a given class in a
large OO program if that class depends on many other classes. 2) Without sufficient “insight”, a
tester may not know where to start testing an OO library. 3) It is extremely costly to construct test
stubs since a tester has to understand the called functions, possibly create and properly initialize
certain objects, and write code to simulate the behaviors and effects of the called functions. 4) It
is impossible to predict all possible uses of a template class and equally impossible to test all
possible uses of a template class. 5) It is difficult to identify and test the effect of polymorphism
and dynamic binding. 6) It is difficult to identify change impact in OO maintenance since the
impact may ripple throughout the OO program through the complex dependencies [kung94g].

3 The Object Relation Diagram

To facilitate a tester to understand the complex relationships among the object classes, we have
developed a test model and a test tool called object relation diagram (ORD). A ORD displays
the various relationships among the object classes, including inheritance, aggregation, association,
instantiation (of a template class), nested, and use (to instantiate a template class). Stating for-
mally, an ORD for an object-oriented program is a multigraph $ORD = (V, E)$, where $V$ is the set
of object classes and $E \subseteq V \times V$ is the set of relationships between the classes. In the following discussion, we use $\text{Class}_A$, $\text{Class}_B$, $\text{Class}_C$, etc. to denote the names of object classes.

The relationships between object classes are identified as follows:

- **Inheritance Relationships.** These relationships are identified according to declarations of the following forms, these declarations are usually found in the header files:

  class $\text{Class}_D$: $\text{Class}_B$

  class $\text{Class}_D$: public $\text{Class}_B$

  class $\text{Class}_D$: protected $\text{Class}_B$

  class $\text{Class}_D$: private $\text{Class}_B$

- **Aggregation Relationships.** These relationships specify that an object class is part of another object class. The class that contains the other class(es) is called the aggregate class. There are three types of aggregations:

  - Automatic aggregations are identified according to declarations of the following forms:

    class $\text{Class}_A$ {

    $\text{Class}_B$ b; // an instance of $\text{Class}_B$ is part of $\text{Class}_A$

    $\text{Class}_C$ c[m]; // an array of instances of $\text{Class}_C$ is part of $\text{Class}_A$

    // ...

    };

  - Static aggregations are identified according to declarations of the following forms:

    class $\text{Class}_A$ {

    static $\text{Class}_B$ b; // a static instance of $\text{Class}_B$ is part of $\text{Class}_A$

    };

  - Manual aggregations are identified according to declarations of the following forms:

    class $\text{Class}_A$ {

    $\text{Class}_B$* b; // a pointer to an instance of $\text{Class}_B$ is part of $\text{Class}_A$

    };


\begin{verbatim}
static Class_C c[m]; // a static array of instances of Class_C is part of Class_A

// ...

};

- Dynamic aggregations are identified according to declarations of the following forms:

  class Class_A {

    Class_B *b; // a dynamic instance of Class_B is part of Class_A

    static Class_C *c[m]; // a dynamic array of instances of Class_C is part of Class_A

    // ...

    Class_A (); // constructor for Class_A }

Class_A::Class_A () {

    // dynamically creating b;

    // dynamically creating c[m];

} 

\end{verbatim}

Note dynamic aggregation requires the aggregate class dynamically creates (i.e., uses the new operator) its member class instances in the free store. If it is not the case, then the relationship is classified as an association relationship.

- **Association Relationships.** There are four types of association relationships:

  - Friend member function associations. These are identified according to declarations of the following forms:

    class Class_A {

    // ...

    friend return_type Class_B::f (...);

    // ...

    \end{verbatim}
- Friend class associations. These are identified according to declarations of the following forms:

class Class_A {

    // ...

    friend class Class_B;

    // ...

};

- Friend operation associations. These are associations between classes through a global function. For example,

class date;

class time {

    // ...

    friend char *time_date (time, date);

    // ...

};

class date {

    // ...

    friend char *time_date (time, date);

    // ...

};

char *time_date (time t, date d) { ... };

- Ordinary associations. These are associations that are established through parameter passing of an instance of one class to a member function of another class.
• **Instantiation and Use Relationships.** These relationships are identified according to the declarations of the following forms, usually found in the source files:

```c
template<class T> class Class_A;
Class_A < Class_C > Class_B;
```

That is, `Class_B` is an instantiation of `Class_A` and `Class_B` uses `Class_C`.

• **Nested Relationships.** These relationships are identified according to the declarations of the following forms, usually found in the header files:

```c
class Class_A {
    ...
    Class_B {
        ...
    }
    ...
}
```

As an example, we show in Figure 1 and ORD for a subset of the InterViews library. We adopt Rumbaugh et al’s notation [rumb91a] due to its wide use, except that we add arrow heads for association relationships. This modification is useful for OO software testing because such directed edges show which classes are dependent on which other classes. The diagram in Figure 1 indicates that `Scene` is a part of `Canvas` and `World` is a subclass of both `Scene` and `Subject`; that is, it is a multiple inheritance. The other relationships can be interpreted similarly.
4 The Test Strategy

We define the test strategy as the order to unit testing and integration testing of the classes in an 
OO program. The test order problem for the classes in an OO program can be stated as finding 
an order to test the classes such that the effort required to construct the test stubs is minimum. 
At the present, our minimum effort test order means that the number of test stubs needed to be 
constructed is minimum. By testing of a class, we mean structure testing, function testing, object 
state testing [kung94c], and/or data flow testing of the member functions of the class [chan93a]. 
Exactly how these testings are performed is beyond the scope of this paper.

The test order also implies effective reuse of previously generated test cases in the new, reusing 
context. In [harr92a], 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 the 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 $ORD = (V, L, E)$ 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 $ORD = (V, L, E)$ is a cyclic digraph, meaning that there exists one or more cycles.
In the first case, the test order is simply the topological sorting of the set of classes using the
dependence relation $R$ (defined in section 3) as the precedence relation [ahoh83a]. The computational
complexity is the number of classes in the OO program since each node needs to be visited only
once. The effort required to construct the test stubs is zero and hence it is minimum.

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

4.1 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 3\(^4\). Note that
a cyclic $ORD$ may have more than one cluster, and a cluster may contain only one vertex (such
clusters are called unit clusters). A cyclic digraph $ORD = (V, L, E)$ can be transformed into an
acyclic digraph $ORD' = (V', L', E')$ in which $V'$ is the set of clusters in the $ORD$, $E'$ is the set of
edges between clusters in the $ORD$, $L'$ is the set of labels on edges in $E'$. It can be proved that
$ORD'$ 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 $ORD'$ 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 temporarily 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.

\[^4\]This is also called strongly connected subgraph in graph theory.
called the minor test order. Thus, a cyclic $ORD$ can be tested first according to the major test order and then the minor test order.

We are now ready to outline the algorithm:

Input: An ORD.

Output: Major and minor test orders for the classes in the ORD.

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

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

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

**Step 4.** For each cycle of the cluster, select and remove an edge(s) 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 straightforward$^5$. 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.

---

$^5$There are algorithms for step 1, we adapt them in section 4.2 for readers who might not be familiar of them.
4.2 Converting a Cyclic ORD to an Acyclic ORD 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 \( ORD = (V, L, E) \). Clearly, if \( u, v \in R^* \) and \( v, u \in R^* \), then \( u \) depends on \( v \) and vice versa. In 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: \( ORD = (V, L, E) \)

Output: an acyclic digraph \( ORD' = (V', L', E') \), where

\[
\begin{align*}
V' &\subseteq \{v' | v' \in 2^V\} \\
L' &\subseteq \{l' | l' \in 2^L\} \\
E' &\subseteq \{< u', v', l' | (u' \times v' \times l') \cap E \neq \emptyset\}\end{align*}
\]

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 \( ORD \) are indexed as \( v_1, v_2, \ldots, v_n \). Any indexing scheme can be used, since it is not essential.

Step 2. Compute \( R^* \). Compute the transitive closure, denoted \( R^* \), for

\[
R = \{< v_i, v_j | v_i, v_j \in V \land (\exists l)(l \in L \land \langle v_j, v_i, l \rangle \in E)\}\]

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

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

insert $v_j$ into the cluster containing $v_i$, i.e., set $[v_i] = [v_i] \cup \{v_j\}$;

mark $v_j$ as *clustered*; and

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:

$$E' = \{< u', v', l' > | u', v' \in V' \land l' \in L' \land (u' \times v' \times l') \cap E \neq \emptyset \}$$

### 4.3 Breaking a Cycle

Given a cyclic *ORD*, 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? Our answer to this question is to break the cycle(s) by temporally removing some of the directed edges. This leads to the second question, that is, which edge(s) are to be removed? Our answer is to remove association edges that will result in the minimum number of test stubs. This is because among the relationships discussed in this paper, association relationships represents the weakest coupling between the related classes. For example, the inheritance and aggregation relationships involve not only control coupling, but also data coupling. The following theorem provides the basis for cycle breaking of association edges:

**Theorem 3.** Every directed cycle of an *ORD* contains an association edge.

The proof of the theorem is lengthy and omitted. The basic idea is that inheritance and aggregation
relationships involve data structure reuse. That is, the derived class in the inheritance case and the aggregate class in the aggregation case reuse and perhaps extend the data structure of the base class or the component class. This is also true for instantiation of a template class using another class. Therefore, if there is a cycle that does not contain an association relation, then the data structures would either extend indefinitely or be brutally converted (i.e., truncated) by the compiler. In either case, the program would not work properly.

After breaking the cycles in each cluster, topological sorting can be applied to produce a test order. Since the generated test order is a topological order, it is possible that two or more vertices may have the same number. This means that the two classes may be tested in either order.

We show in Figure 2 an ORD diagram along with the class test order for the subset of the InterViews library in Figure 1.

5 Experience

Several years ago, some of us were involved in the development of a commercial product using C++. We were astonished by the lack of systematic OO testing methods and supporting tools to help a tester to carry out the testing process. We discovered that the traditional testing techniques and methods were inadequate since these techniques and methods did not address the complexity of an OO program. When we started to test one class, we soon found out that one had to trace and understand many other classes in order to construct the test cases and test stubs. We realized that an OO test model to provide a high level abstraction of the OO program was needed. This model should help a tester in understanding the complex relations and coping with the complex dependencies among the various components of the OO program. The development of the ORD and the test order tool was motivated by this experience. After applying the tools to application
Figure 2: The test order for the subset of InterViews
programs, we feel that the tools help the tester understand the structures of, and relations between the components of an OO program. They provide the tester a systematic method to perform OO testing. In particular, it assists the tester to find better test strategies to reduce testing effort.

To illustrate the significance of the tools, we show in Figure 3 the number of test stubs needed for unit testing of the InterViews library when a randomly generated test order is used. The total number of stubs required is 400 (i.e., 3.27 per class × 122 classes). If we use the result obtained in our experiment mentioned earlier, each stub requires 0.79 person-hours to prepare, then 400 stubs would require 316 person-hours or almost 8 person-weeks. In comparison, if the optimal test order is used, then it requires only 8 test stubs. That is, 93% saving is achieved. Using the test order still requires 8 test stubs because there are cyclic dependencies among the classes. The weakest dependency in a cycle must be simulated by a test stub(s) so that the remaining dependencies become acyclic and a (topological) test order can be defined for the classes.

Integration testing is not an easy task. Traditionally, there are several integration strategies, including top-down, bottom-up, and sandwich approaches [beiz90a]. It has been recognized that a
Figure 4: Test stubs required for random integration testing sequences

A bottom-up strategy is preferred for OO programs. However, how to conduct bottom-up OO testing is still an open problem. One notable contribution is [harr92a] in which a bottom-up methodology was proposed for testing the inheritance hierarchy of OO programs. Jorgensen and Erikson [jorg94a] proposed and outlined a five level testing approach that also advocates bottom-up integration. Our experience indicates that the ORD and the test order are not only useful for conducting class unit testing but also provide a detailed road map for conducting integration testing. That is, after conducting class unit testing, the classes are to be integrated according to the test order. In this way, the effort required to construct test stubs and test drivers can be reduced to a minimum.

Consider, for example, the integration testing of the InterViews library. Figure 4 shows the statistics of test stubs required for 100 randomly generated integration sequences. We see that the average number of stubs required is 191.88 per sequence. If each stub requires 0.79 person-hours to prepare, then it would require about 152 person-hours or almost 4 person-weeks⁶. When using the test order, then only 8 test stubs are needed. That is, 96% savings can be achieved.

⁶Fewer test stubs are required than the random unit test sequence because the result in Figure 4 is an average over 100 sequences and the result in Figure 3 is for one random sequence, which may happen to be a costly one.
6 Conclusions

We have discussed two of the problems in testing an OO program. We show that an object relation diagram is useful for computing an optimal test order for unit testing and integration testing of OO programs. The ORD tool has also been used in the regression testing phase to identify the changes and change impact [kung94g]. The tools have been shown to many industrial visitors and ported to a number of sponsoring companies. The feedback from industry is very promising. We are now refining the test order algorithm to provide more accurate estimates of the effort required to generate the test stubs.

7 Acknowledgment

The material presented in this paper is based on work supported by the Texas Advanced Technology Program (Grant No. 003656-097), BBN Systems and Technologies, Electronic Telecommunications Research Institute (ETRI) of Korea, Fujitsu Network Transmission Systems, Inc., Hewlett-Packard Company, IBM, Office of Naval Research, Sun Microsystems Computer Company, and the Software Engineering Center for Telecommunications at UTA.

Thanks are due to the OOT project team members for developing the tools and collecting the various statistics.

8 References


