Change Impact Identification in Object Oriented Software Maintenance

D. Kung, J. Gao, P. Hsia, F. Wen
Computer Science Engineering
Univ. of Texas at Arlington
P. O. Box 19015
Arlington, TX 76019-0015
Tel: (817) 273-3627
Fax: (817) 273-3784
Email: kung@cse.uta.edu
Y. Toyoshima, and C. Chen
Fujitsu Network Transmission Systems, Inc.

Abstract

Types of code changes in an object oriented library are described. A formal model for capturing and inferencing on the changes to identify affected classes is described. The model consists of three types of diagrams: the object relation diagram (ORD), the block branch diagram (BBD), and the object state diagram (OSD). An ORD describes the inheritance, aggregation, and association relationships between the classes of a C++ library. A BBD describes the control structure and interfaces of a member function. And an OSD describes the state behavior of a class. Unlike in modeling, these diagrams are automatically generated from code and facilitate understanding and changing a C++ library. An OO software maintenance environment that implements the research result is described. Our experience with the environment prototype shows promising results.

Key words and phrases: software maintenance, object oriented programming, change analysis, impact identification, regression testing, environment, tool

1 Introduction

One important activity of software maintenance is regression testing, which ensures that the modified software still satisfies its intended requirements. To save effort, regression testing should retest only those parts that are affected by the modifications.

In traditional function-oriented programming, only control dependencies exist between the modules; and hence, it is relatively easy to identify the affected modules. In the object-oriented (OO) paradigm, a number of new features is supported, such as encapsulation, information hiding, inheritance, aggregation, polymorphism, and dynamic binding. These new features introduce new problems in the maintenance phase, including difficulty of identifying the affected components when changes are made.
Encapsulation and information hiding imply the so-called “delocalized plan” [solo88a], in which several member functions from possibly several object classes are invoked to achieve an intended functionality. This phenomenon means that changes to a member function of a class may affect many classes. Inheritance and aggregation imply structure and state dependent behavior reuse, i.e., the data members, function members, and state dependent behavior of a class are re-used by another class. Thus, there are data dependencies, control dependencies, and state behavior dependencies between the two classes. Moreover, since the inheritance and aggregation relations are transitive relations, the above dependencies also are transitive dependencies. Polymorphism and dynamic binding imply that objects may take more than one form, and which form an object assumes is unknown until run time. This makes the identification of the affected components much more difficult.

The maintenance complications introduced by the OO features can be summarized as follows: 1) although it is relative easy to understand most of the data structures and member functions of the object classes, understanding of the combined effect or combined functionality of the member functions is extremely difficult; 2) the complex relationships between the object classes make it difficult to anticipate and identify the ripple effect\(^1\) [gane82a] of changes; 3) the data dependencies, control dependencies, and state behavior dependencies make it difficult to prepare test cases and generate test data to “adequately” retest\(^2\) the affected components; 4) the complex relations also make it difficult to define a cost-effective test strategy to retest the affected components.

In an attempt to solve above problems, the Software Engineering Center for Telecommunications at UTA and Fujitsu Network Transmission Systems, Inc. have undertaken a major effort in the last two years to develop a methodology for OO software testing and maintenance. The results obtained so far include: 1) a reverse engineering approach for design recovery of C++ programs [kung93b]; 2) a three level schema and algorithms for data flow testing of OO programs [chan93a]; 3) definition and identification of class firewalls and a test strategy for regression testing of C++ programs [kung94a]; 4) a specification-based method for object state testing [kung94b]; and 5) a program-based method for object state testing [kung94c]. Part of these results and other results have been implemented in an integrated testing and maintenance environment [kung93g] [song93a].

This paper discusses types of changes that can be made to an OO library. It also describes a method for identifying the affected classes due to structure changes to an object class library. The method is based on a reverse engineering approach to extract the classes and their interrelationships. This information is represented in a multigraph, which is used to automatically identify the changes and the effects of changes. The method has been implemented in the integrated testing and maintenance environment. The architecture and functionality of the relevant part will be presented.

The organization of this paper is as follows. Section 2 gives a brief review of related work on maintenance of conventional as well as OO software. In section 3, we discuss types of changes, and change identification. A formal model is presented to facilitate change identification and impact identification, which is described in section 4. In section 5, we describe a support system for OO testing and maintenance. In section 6, we report our experience on OO software maintenance and in section 7 we present the conclusions and future work.

\(^1\)The ripple effect refers to the phenomenon that changes made to one part of a software system ripple throughout the system.

\(^2\)We use this to loosely mean retesting the software with certain degree of confidence. We choose not to give a formal definition of adequacy in this paper.
2 Related Work

Hartmann and Robson examined several regression testing strategies, including methods for capturing the program portion which may be affected by maintenance modifications to a conventional program [hart89a]. A similar study was conducted by Leung and White [leun91a] using a formally defined cost model. Laski and Szumer described an algorithm for identifying the affected parts in conventional program maintenance [lask92a]. The algorithm is based on differentials between the control flow graphs for the original program and the modified program. In [leun90a] impact of data and function changes is addressed using a dynamic approach.

Some conventional program maintenance systems have been reported in the literature. The $R^n$ environment [hood84a] and the VIFOR (Visual Interactive FORtran) [raj90a] were developed for FORTRAN programs, the MasterScope [teit81a] for Interlisp, and the CIA (C Information Abstractors) [chen90a] for C. These systems provide editing, browsing, and database supports to a maintainer. In particular, the VIFOR system also provides graphical display and transformations between the textual representation and the graphical representation.

Wilde and Huit [wild92a] analyzed problems of dynamic binding, object dependencies, dispersed program structure, control of polymorphism, high level understanding and detailed code understanding. The authors then provided a general list of recommendations, including the use of dependency analysis [wild89a] and clustering methods [choi90a] [lius90a] [schw89a] [selb88a], for possible tool support.

Crocker and Mayrauser addressed problems relating to class hierarchy changes, class signature changes, and polymorphism [croc93a]. The authors then proposed a list of tools to help solve some of the problems. The tools provide information collection, storage, analysis, inference, and display capabilities.

An early system for maintaining C++ programs was reported by Samehncerg in [same90a]. The system utilizes the inheritance relation and containment relations (e.g., a class is contained in a file, or a method belongs to a class, etc.) to provide text based browsing facilities to an OO software maintainer.

The C++ Information Abstractors [gras90a] use program analyzers to extract cross reference information and store the information in a database. A maintainer can query the data base to obtain the desired knowledge to maintain a C++ program.

Lejteker, Meyers and Reiss discussed the difficulty of maintaining an OO software system due to the presence of inheritance and dynamic binding [lejt92a]. The authors then described the XREF/XREFDB prototype system that provides text editing and relational data base querying support to facilitate OO software maintenance. A similar system is described in [obri87a].

Our system is similar in many aspects to the above systems. It uses program analyzers to collect information and stores the information in a data base. It provides both graphical and textual display and browsing, whereas most of the existing systems provide only textual display and browsing (with VIFOR as an exception). It is capable of automatically identifying the changes to an OO program and deriving the affected parts from the changes. Another difference is that our system is integrated with testing capabilities to facilitate regression test case and test data reuse and generation, result analysis, and report generation.
3 Change Identification

One of the major difficulties in software maintenance is to identify changes and their impact automatically since it is very difficult to keep track of the changes when a software system is modified extensively by several persons. This capability becomes even more crucial when the modifications are performed by one group of persons and regression testing is performed by another group of persons.

In this section, we first discuss the different types of code changes in section 3.1. We then present, in section 3.2, a formal model for capturing such changes Finally, we describe in the remaining sections how to identify the various types of code changes.

3.1 Types Of Code Changes

Figure 1 provides a classification of code changes in an OO class library. These change types are explained as follows:

- **Data change.** Any datum (i.e., a global variable, a local variable, or a class data member) can be changed by updating its definition, declaration, access scope, access mode and initialization. In addition, adding new data and/or deleting existing data are also considered as data changes.

- **Method change.** A member function can be changed in various ways. Here we classify them into three types: component changes, interface changes, and control structure changes.
  Component changes include: 1) adding, deleting, or changing a predicate, 2) adding, deleting a local data variable, and 3) changing a sequential segment.
  Structure changes include: 1) adding, deleting, or modifying a branch or a loop structure, and 2) adding, or deleting a sequential segment.
  The interface of a member function consists of its signature, access scope and mode, its interactions with other member functions (for example, a function call). Any change on the interface is called an interface change of a member function.

- **Class change.** Direct modifications of a class can be classified into three types: component changes, interface changes and relation changes. Any change on a defined/redefined member function or a defined data attribute is known as a component change. A change is said to be an interface change if it adds, or deletes a defined/redefined attribute, or changes its access mode or scope. A change is said to be a relation change if it adds, or deletes an inheritance, aggregation or association relationship between the class and another class.

- **Class library change.** These include: 1) changing the defined members of a class, 2) adding, or deleting a class and its relationships with other classes, 3) adding, or deleting a relationship between two existing classes\(^3\), 4) adding, or deleting an independent class.

\(^3\)Changing a relationship R1 (between two classes) into a relationship R2 is considered as deleting R1 and adding R2.
<table>
<thead>
<tr>
<th>Components</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Method</td>
<td>interface changes</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>structure changes</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>component change</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>component change</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>relationship change</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>component change</td>
</tr>
<tr>
<td>Library</td>
<td>relationship change</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Different Types Of Code Changes
3.2 A Formal Model

Our formal model was originally developed for capturing and representing the complex relationships and interdependencies between the various parts of a C++ program. The model consists of three types of diagrams, i.e., Object Relation Diagrams (ORD), Block Branch Diagrams (BBD), and Object State Diagrams (OSD). An ORD facilitates the understanding of the inheritance, aggregation, and association relationships\(^4\) between the classes. These relationships are extracted from code and displayed graphically. Figure 2 shows a screen dump of an ORD for a subset of the InterViews library. We postpone the formal definition of ORD and explanation of the screen dump til section 3.6.

A BBD facilitates the understanding of the member functions and their relationships to the global data, class data, and other member functions. Figure 3 shows a BBD for a member function of the InterViews library. The various components of a BBD is explained as follows:

- the large window displays the BBD body, denoted B; it encapsulates the program graph for the member function\(^5\).
- the upper left window displays the global and class data that are used by the member function; this is denoted by Du;
- the upper right window displays the input/output parameters, denoted P, of the member function;
- the bottom left window displays the global and class data that are defined (i.e., updated) by this member function; this is denoted by Dd;
- the bottom right window displays functions that are called by this member function; this is denoted by Fe;

Formally, the block branch diagram for a member function \(f\) is a quintuple

\[
BBD_f = (D_u, D_d, P, F_e, B)
\]

where the components are as defined above. When no confusion can arise, we will omit the subscript \(f\) from \(BBD_f\).

A BBD body is formally defined by a directed graph \(B = (V, E)\), where \(V\) denotes the set of program graph vertices and \(E \subset V \times V\) the directed edges representing the control flows. As usual, \(B\) satisfies the following conditions: 1) there is exactly one starting vertex (which has indegree zero) and one final vertex (which has outdegree zero); 2) all the other vertices have indegree one and outdegree either one or two; 3) except the starting and final vertices, each vertex also satisfies the following conditions: a) if it has outdegree one, then it represents either a function call or a sequence of simple statements; b) if it has outdegree two, then it represents a decision vertex for a simple condition; 3) every vertex of \(B\) occurs on some path from the starting vertex to the final vertex. For more details the reader is referred to \([\text{kung93b}]\).

\(^4\)The meanings of these relationships are similar to those used in OO modeling, but there are subtle differences, as we have addressed in \([\text{kung94a}]\).

\(^5\)The program graph can be used, among others, to generate basis path test cases and test data \([\text{beiz90a}]\). However, it is beyond the scope of this paper to explore this issue.
Figure 2: An ORD for a part of the InterViews Library
Figure 3: A BBD for a member function in the InterViews Library
The OSD is not closely related to the work in this paper; and hence, it is omitted. The interested reader is referred to [kung93b] [kung94b] [kung94c].

3.3 Data Change Identification

Data change identification is easy since the needed information is captured by the BBD’s (and the internal representation) for the member functions\(^6\). In particular, information about each data item’s access scope, type, access mode, update set (i.e., functions that define the data item), and use set (i.e., functions that use the data item). To identify data change, this information is compared with the information for the original software. If any of the above information is different, the corresponding type of change is identified.

3.4 Method Change Identification

We will use method and member function interchangeably. Let BBD = \((D_u, D_d, P, F_c, B)\), and BBD’ = \((D_u’, D_d’, P’, F_c’, B’)\) be the BBDs for a member function C::f(...) and its modified version C::f’(...) respectively. Recall that \(B = (V, E)\) (or \(B’ = (V’, E’))\) is a directed graph which represents the control structure of C::f(...) (or C::f’(...)). Method structure and/or component changes are identified as follows:

1. if \((V – V’) \neq \emptyset\) then any \(v \in (V – V’)\) is a deleted block node.
2. if \((V’ – V) \neq \emptyset\) then any \(v \in (V’ – V)\) is an added block node.
3. if \((E – E’) \neq \emptyset\) then any \(e \in (E – E’)\) is a deleted control edge.
4. if \((E’ – E) \neq \emptyset\) then any \(e \in (E’ – E)\) is an added control edge.

A member function interface change is identified as follows:

1. if \((D_u – D_u’) \neq \emptyset\), then some data uses are removed.
2. if \((D_u’ – D_u) \neq \emptyset\), then some data uses are added.
3. if \((D_d – D_d’) \neq \emptyset\), then some data definitions are removed.
4. if \((D_d’ – D_d) \neq \emptyset\), then some data definitions are added.
5. if \((F_c – F_c’) \neq \emptyset\), then some function calls are removed\(^7\).
6. if \((F_c’ – F_c) \neq \emptyset\), then some function calls are added.

\(^6\)Functions not belong to any class are treated as member functions of a dummy system class in our approach.
\(^7\)Signature change is treated as deleting and then adding a function.
3.5 Class Change Identification

A class is a pair $C = (D_{def}, F_{def})$, where $D_{def}$ is a set of defined/redefined data attributes, $F_{def}$ is a set of defined/redefined member functions.

Let $C' = (D'_{def}, F'_{def})$ be a modified version of a class $C$. Then class code change is identified as follows:

- if $(D_{def} - D'_{def}) \neq \emptyset$, then any $d \in (D_{def} - D'_{def})$ is a deleted data attribute.
- if $(D'_{def} - D_{def}) \neq \emptyset$, then any $d \in (D'_{def} - D_{def})$ is an added data attribute.
- if any $d \in D'_{def} \cap D_{def}$ is changed, then a residual data attribute is changed.
- if $(F_{def} - F'_{def}) \neq \emptyset$, then any $f$ in $(F_{def} - F'_{def})$ is a deleted member function.
- if $(F'_{def} - F_{def}) \neq \emptyset$, then any $f$ in $(F'_{def} - F_{def})$ is an added member function.
- if any $f \in F'_{def} \cap F_{def}$ is changed, then a residual defined/redefined member function is changed.

3.6 Class Library Change Identification

A class library $L$ is a collection of ORDs. An ORD is an edge labeled directed graph $ORD = (V, L, E)$, where $V$ is the set of nodes representing the object classes, $L = \{I, Ag, As\}$ is the set of edge labels (for inheritance, aggregation, and association), and $E = E_I \cup E_{AG} \cup E_{AS}$ is the set of edges. For a detailed definition and how to reverse code to yield an ORD, the reader is referred to [kung93b]. As mentioned earlier, Figure 2 shows the screen dump of an ORD for part of the InterViews library. In the figure, the inheritance and aggregation relationships are shown using Rumbaugh et al’s notation, while association is shown using directed arcs. The figure says that Canvas is associated with CanvasRep, and hence dependent on CanvasRep. ControlState and Scene are parts of Canvas, and hence Canvas is dependent on ControlState and Scene. The figure also shows that World is a derived class of Canvas, and hence, dependent on Canvas.

Figure 2 also shows a window containing the header file information for the Scene class. This information is obtained by clicking on the rectangle labeled by “Scene”. From the header file information, a maintainer can request the system to display the details of a member by clicking on that member. But this latter capability is not shown in Figure 2.

Modifications to a library can be classified into three basic cases, i.e., adding an ORD, deleting an ORD, and changing an ORD\(^8\). In the first two cases, there is no impact to the other ORD’s, therefore, we will consider only the last case. An ORD can be changed in several ways: changing the defined members of a class, adding/deleting a relation between two existing classes, and adding/deleting a class and its relations. Change identification for a single class has been discussed in the previous subsection. Here, we focus on structure change of a class library.

Let $ORD = (V, L, E)$ and $ORD' = (V', L', E')$ be the ORD’s for two versions of the same software. A structure change in an ORD is identified as follows:

- if $(V' - V) \neq \emptyset$ then any $v \in (V' - V)$ is an added class node.

---

\(^8\)Note an isolated class is an ORD.
Figure 4: Different Structural Changes In An ORD

- if \((V - V') \neq \emptyset\) then any \(v \in (V - V')\) is a deleted node.
- if \((E' - E) \neq \emptyset\) then any \(e \in (E' - E)\) is an added new edge.
- if \((E - E') \neq \emptyset\) then any \(e \in (E - E')\) is a deleted edge.
- if any \(v\) in \(V \cap V'\) is changed, then a residual class is changed.

To facilitate understanding, Figure 4 shows the different types of structure changes for an ORD. Figures 4(a) and (b) show the two cases in which a new class node is added into the original ORD by inserting a new relation with an residual class. Two cases of a class deletion are given in Figures 4(c) and (d). Notice that if a class node is removed from an ORD, then all relation edges between the class node and other classes must be deleted. In Figure 4(e), an existing relation edge is removed from two residual classes. In Figure 4(f), a new relation edge between two residual classes is added.

\(^9\text{An independent single class in a class library is considered as an ORD. Thus, adding (or deleting) an independent single class adds (or deletes) an ORD.}\)
4 Change Impact Identification

After change identification, it is very important to detect the ripple effects of the changes. Because these changes may affect a software in different aspects, including functions, structures, behaviors and performance. Clearly understanding these effects not only reduce the cost of software maintenance, but also save the regression test effort. For example, if we can find all the affected components in the give software, the testers only need to retest these components instead of all of the components in the software. Thus, it is very important to identify and enclose all the affected components after changing a software.

Change impact identification includes data change impact identification, method change impact identification, class change impact identification, and class library change (i.e., class relation change) impact identification. Existing results, e.g., [leun90a], can be used to identify data and method change impacts. Class change impact has been addressed in [kung94a]. Therefore, we focus on class library change impact identification in this paper.

4.1 Class Library Change Impact Identification

The major task of class library change impact identification is to find all the affected classes due to changes of interclass relationships. We use a class firewall concept introduced in [kung94] to enclose all affected classes. The computation of a class firewall is based on the ORDs for the class library.

As shown in Figure 4, there are six types of structure changes in an ORD. Taking into consideration the differences in treating the inheritance, aggregation and association relationships, the impact on a residual class is identified as follows:

(a) Adding a new “superclass”:
   i. If the new edge is an inheritance edge, then it expands A’s members. According to Weyuko’s antidecomposition axiom\(^{10}\), we should retest these new inherited members in the context of class A [pbry70]. In addition, integration testing of class A’s new inherited members and its other members is needed to make sure that they work well together.
   ii. If the edge is an inheritance or aggregation edge, then it expands A’s state space in terms of state behavior (see Figure 4.1a). That is, its states and transitions are extended by its inherited members from class B. According to Weyukos’s anticomposition axiom\(^{11}\), we should retest the state behaviors of class A.
   iii. If the edge is an association edge, then at least one method of class A must be changed, and hence, A is affected in this sense.

(b) Adding a new dependent class.
   Class A is not affected and hence does not need to be retested.

(c) Deleting a “superclass”.\(^{10}\)

\(^{10}\) There exists a program P and component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q. [weyu88a]

\(^{11}\) There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P; Q[P;Q is the composition of P and Q. [weyu88a]
This changes class A by removing class B from class A’s superclass or component class list:

i. If the edge is an inheritance edge, then it reduces the inherited members. This only affects class A’s test cases which related to those removed members.

ii. If the edge is an inheritance or aggregation edge, then it reduces the states and transitions of class A (see Figure 4.1b). According to Weyko’s antidecomposition axiom, we should retest the state behaviors of the modified class A.

iii. If the edge is an association edge, then at least one method of class A must be changed, and hence, A is affected in this sense.

(d) Deleting a dependent class.
Class A is not affected.

(e) Deleting a relationship between existing classes.
This is similar to (c) above.

(f) Adding a relationship between existing classes.
This is similar to (a) above.

We have discussed the different types of structure changes of an OO library. We showed some of the changes affect a residual class A even though its defined attributes are not changed at all. Thus, class A should be retested. A residual class like class A is called an addition affected class if it is directly affected by any edge addition (or class addition) in an ORG (cases (a) and (f) above). Similarly, a residual class is called a deletion affected class if it is directly affected by any edge deletion (or class deletion) in an ORG (cases (c) and (e) above).
4.2 Class Firewall Generation

The basic idea of computing a class firewall is described in [kung94]. A binary relation R, derived from an ORG = (V, L, E), is introduced to compute the class firewall.

\[ R = \{ < C_j, C_i > | C_i, C_j \in V \land < C_i, C_j, l > \in E \} \]

where R is the dependence relation which defines the dependence between classes, according to the inheritance, aggregation, and association relations.

In [kung94], we have proved that for an ORG, if the modification a class library changes a class C without altering the structure of the ORG, then a class firewall for C, denoted \( \text{CCFW}(C) \), can be computed as follows:

\[ \text{CCFW}(C) = \{ C_j | < C, C_j > \in R^* \} \]

where \( R^* \) is the transitive closure of R. That is, if \( < C_i, C_j > \in R \) and \( < C_j, C_k > \in R^* \), then \( < C_i, C_k > \in R^* \). The transitive closure of R can be computed by the famous Warshall algorithm [aho83].

Let \( L \) be a class library, and \( L' \) its modified version. Assume that ORD = \( (V, L, E) \) be an ORG in \( L \), and ORD' = \( (V', L', E') \) its modified version. We define a binary relation \( R_r \) as

\[ R_r = R \cap (V \times V) \cap (V' \times V') \]

where \( \times \) is the Cartesian product operation. In other words, \( R_r \) is the relation which defines the dependencies between the residual classes.

The class firewalls are generated as follows:

- The class firewall for a changed class C is computed as follows:
  \[ \text{CCFW}(C) = \{ C_j | < C, C_j > \in R^*_r \} \]

- The class firewall for addition affected classes is computed as follows:
  \[ \text{CCFW} = \{ C_j | (\exists C) (\exists k) ((< C_j, C, k > \in E' - E) \land < C, C_j > \in R^*_r) \} \]

  where \( R' \) is the dependence relation for the modified ORD.

- The class firewall for deletion affected classes is computed as follows:
  \[ \text{CCFW} = \{ C_j | (\exists C) (\exists k) ((< C_j, C, k > \in E - E') \land < C, C_j > \in R^*_r) \} \]

5 Support System

The objective of a software maintenance system is to help a software maintainer in: understanding a given software, identifying code changes, supporting software updates and enhancement, detecting change effects. Therefore, any software maintenance system has to fulfill four important requirements: 1) it has to be able to provide the various information about the software, including its structures, and the interfaces and relationships between different components at the different levels; 2) it has to provide an efficient and user-friendly interface to present the various information about software and support maintenance activities; 3) it has to be able to identify code changes between different software versions and their effects.
Figure 5: The Architecture Of OOTME

Figure 5 depicts the architecture of an OO testing and maintenance environment we are currently developing.

The components of the environment are described as follows:

- **GUI**: The GUI is constructed based on Motif and X window software, it is user-friendly and easy to use.
- **Parser**: There are three different parsers: the ORD parser, the BBD parser, and the OSD parser. The user can use them to extract information from a class library.
- **Displayer**: There are three Displayers: the ORD Displayer, the BBD Displayer, and the OSD Displayer. The user can use them to display the ORDs, BBDs, and OSD’s, respectively.
- **Change identifier**: The change identifier can be used to find the code changes between two different versions of the same class library. It can also be used to automatically identify the impact of a planned code change to help a maintainer to determine whether the change shall be carried out.
- **Firewall generator**: The firewall generator can be used to identify a class firewall to enclose all the possible affected classes in a class library when it is modified.
- **Regression test tool**: The regression test tool consists of: test strategy generator, test case generator. The test strategy generator produces a cost-effective test order for each class in the library [kung94]. This test order will be used as a re-test sequence in class unit testing, and re-integration testing. The test case generator is used to generate new test cases.

We have applied this tool to many applications, including the Interviews library. Figure 6 shows an example listing of the firewall information for a subset of the InterViews library. We have also compared the two versions (3.0 and 3.1) of the Interviews library using the OOTME. It lists all of the differences between these two versions, including 68 added classes, 76 deleted classes, and
Figure 6: Change impact identification
46 residual classes (reused classes). In addition, it also identified 26 changed residual classes, 16 affected classes and 4 unaffected classes. The detailed result is given in tables 1 – 3.

<table>
<thead>
<tr>
<th>class name</th>
<th>class name</th>
<th>class name</th>
<th>class name</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveHandler</td>
<td>Adjustable</td>
<td>AllocationInfo</td>
<td>AllocationTable</td>
<td>AllocationTableImpl</td>
</tr>
<tr>
<td>BoxImpl</td>
<td>CoordinateSpace</td>
<td>DebugGlyph</td>
<td>Dialog</td>
<td>DialogHandler</td>
</tr>
<tr>
<td>DialogKitImpl</td>
<td>Drag</td>
<td>DragZone</td>
<td>DragZoneSink</td>
<td>FieldButton</td>
</tr>
<tr>
<td>FieldEditorImpl</td>
<td>FieldStringEditor</td>
<td>FileBrowserImpl</td>
<td>FileChooserImpl</td>
<td>FontBoundingBox</td>
</tr>
<tr>
<td>GLContext</td>
<td>Input Handler</td>
<td>InputHandlerImpl</td>
<td>LayoutKit</td>
<td>LayoutKitImpl</td>
</tr>
<tr>
<td>LayoutLayer</td>
<td>MFDialogKitImpl</td>
<td>MFKitForeground</td>
<td>MFKitFrame</td>
<td>MFKitImpl</td>
</tr>
<tr>
<td>MKKitInfo</td>
<td>MKFKitMenuItem</td>
<td>MKItem</td>
<td>MKItem</td>
<td>MKItem</td>
</tr>
<tr>
<td>OL_Anchor</td>
<td>OL_Button</td>
<td>OL_Cable</td>
<td>OL_Channel</td>
<td>OL_CheckBox</td>
</tr>
<tr>
<td>OL_CheckMark</td>
<td>OL_DragBox</td>
<td>OL_Elevator</td>
<td>OL_ElevatorGlyph</td>
<td>OL_FieldEditor</td>
</tr>
<tr>
<td>OL_Frame</td>
<td>OL_Gauge</td>
<td>OL_Indicator</td>
<td>OL_MenuMark</td>
<td>OL_Mover</td>
</tr>
<tr>
<td>OL_Pushpin</td>
<td>OL_PushpinLook</td>
<td>OL_Scrollbar</td>
<td>OL_Setting</td>
<td>OL_Slider</td>
</tr>
<tr>
<td>OL_Specs</td>
<td>OL_Stepper</td>
<td>OL_Tick</td>
<td>OL_ToLimit</td>
<td>Observable</td>
</tr>
<tr>
<td>Observer</td>
<td>WidgetKitImpl</td>
<td>WidgetKitOverlay</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: New added classes in InterViews3.1

<table>
<thead>
<tr>
<th>class name</th>
<th>class name</th>
<th>class name</th>
<th>class name</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationWindow</td>
<td>BMargin</td>
<td>BoxAllocation</td>
<td>BoxComponent</td>
<td>BreakSet</td>
</tr>
<tr>
<td>Center</td>
<td>DeckInfo</td>
<td>FixedSpan</td>
<td>HCenter</td>
<td>HGlue</td>
</tr>
<tr>
<td>HMargin</td>
<td>HRule</td>
<td>Hit Target</td>
<td>HitTargetList</td>
<td>IconWindow</td>
</tr>
<tr>
<td>LMargin</td>
<td>LRBox</td>
<td>Listener</td>
<td>ManagedWindow</td>
<td>Margin</td>
</tr>
<tr>
<td>OptionDesc</td>
<td>Overlay</td>
<td>PSFont</td>
<td>PFontImpl</td>
<td>Page</td>
</tr>
<tr>
<td>FagelInfo</td>
<td>Patch</td>
<td>Pattern</td>
<td>PointerHandler</td>
<td>PopupWindow</td>
</tr>
<tr>
<td>PossibleHitTarget</td>
<td>Printer</td>
<td>PrinterInfo</td>
<td>PrinterRep</td>
<td>PropertyData</td>
</tr>
<tr>
<td>RMargin</td>
<td>Raster</td>
<td>Regexp</td>
<td>ReqErr</td>
<td>Resource</td>
</tr>
<tr>
<td>ResoncElmImpl</td>
<td>Rule</td>
<td>Sensor</td>
<td>Session</td>
<td>SessionIOHandler</td>
</tr>
<tr>
<td>SessionRep</td>
<td>Shadow</td>
<td>SimpleComposer</td>
<td>Stencil</td>
<td>Style</td>
</tr>
<tr>
<td>StyleAttribute</td>
<td>StyleRep</td>
<td>StyleWildcard</td>
<td>StyleWildcardInfo</td>
<td>StyleWildcardMatchQuality</td>
</tr>
<tr>
<td>Superpose</td>
<td>TBBbox</td>
<td>TIFFRaster</td>
<td>TIFFRasterImpl</td>
<td>TMargin</td>
</tr>
<tr>
<td>Target</td>
<td>ToXCompositor</td>
<td>Tile</td>
<td>TileReversed</td>
<td>TopLevelWindow</td>
</tr>
<tr>
<td>TransformSetter</td>
<td>Transformer</td>
<td>TransientWindow</td>
<td>VCenter</td>
<td>VGlue</td>
</tr>
<tr>
<td>VMargin</td>
<td>VRule</td>
<td>ValueString</td>
<td>Window</td>
<td>World</td>
</tr>
<tr>
<td>XYMarker</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Deleted classes from InterViews3.0

<table>
<thead>
<tr>
<th>class name</th>
<th>class name</th>
<th>class name</th>
<th>class name</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C)Aggregate</td>
<td>(C)Allocation</td>
<td>(C)Background</td>
<td>(C)Bitmap</td>
<td>(C)Border</td>
</tr>
<tr>
<td>(C)Box</td>
<td>(C)Break</td>
<td>(C)Brush</td>
<td>(C)Canvas</td>
<td>(C)Color</td>
</tr>
<tr>
<td>(C)Composition</td>
<td>(C)Deck</td>
<td>(C)Display</td>
<td>(C)Event</td>
<td>(C)Extension</td>
</tr>
<tr>
<td>(C)Font</td>
<td>(C)FontFamily</td>
<td>(C)Glyph</td>
<td>(C)Handler</td>
<td>(C)Hit</td>
</tr>
<tr>
<td>(C)HitImpl</td>
<td>(C)Image</td>
<td>(C)LRComposition</td>
<td>(C)Monoglyph</td>
<td>(C)Requisition</td>
</tr>
<tr>
<td>(C)TBComposition</td>
<td>(A)AggregateInfo</td>
<td>(A)Align</td>
<td>(A)Character</td>
<td>(A)CompositionComponent</td>
</tr>
<tr>
<td>(A)Control</td>
<td>(A)Discretionary</td>
<td>(A)Glue</td>
<td>(A)Group</td>
<td>(A)HStur</td>
</tr>
<tr>
<td>(A)LRRMarker</td>
<td>(A)Label</td>
<td>(A)Layout</td>
<td>(A)ShapeOf</td>
<td>(A)Space</td>
</tr>
<tr>
<td>(A)Strut</td>
<td>(A)YStrut</td>
<td>(U)Allocation</td>
<td>(U)ArrayCompositor</td>
<td>(U)Compositor</td>
</tr>
<tr>
<td>(U)Requirement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Reused classes in InterView3.1

17
6 Experience

Aids in understanding an OO software system, anticipating and identifying the effect of change, and facilitating regression testing are desirable capabilities of an OO software maintenance environment.

Our experience indicates that it is extremely time consuming and tedious to test and maintain an OO software system. This problem becomes even acute when documentation is either missing or inadequate. Consider for example the InterViews library originally developed by Stanford University. An early version of the library consisted of 146 files, more than 140 classes, more than 400 relationships, and more than 1,000 member functions. We felt difficult to obtain a high level understanding of the classes, member functions, and their relationships. Our initial experiment showed that it takes in average two hours for a C++ programmer to understand a small member function (ranging from a few lines to less than 20 lines) that invokes other member functions. Without tool support, it is almost impossible to anticipate and identify the effect of change because a class can be instantiated by another class which can then use the capabilities of the former. To ensure quality of our products, we often had to retest the entire library when major changes were made.

Software engineers in the industry consider the system useful for testing, maintenance, and reengineering. The application of the system to the InterViews library, an elevator library, and a PBX program shows that it facilitates understanding, test ordering [kung94a], test case generation (for which automatic support will be available in the future), and effort estimation (in terms of extended cyclomatic complexity and COCOMO model [boeh84a]).

The application of the automatic change and change effect identification capabilities to the InterViews library has produced promising results. However, our experience is too limited to draw any conclusions. We anticipate that these capabilities will be particularly useful in the maintenance phase. Without tool support, one has to document what have been changed and identify the impact of the changes according to one’s knowledge of the system. This is both time consuming and inaccurate. The tool will reduce costs and improve productivity in documentation and regression testing by automatically identifying the changes and their impact.

Although the system is still in its prototyping stage, several companies have expressed interests in experimenting with the system and we are porting it to more companies. Since we did not emphasize portability during prototyping, porting to other platform and operating system is not easy.

7 Conclusion and Future Work

We have described the various types of code changes of an OO system and a formal model for capturing and inferencing on the changes to identify affected components.

The model and the inference capabilities have been implemented in a tool prototype. Experience with an earlier version of the tool shows promising results.

As reported elsewhere [kung94a], the changed and affected classes can be tested in a cost-effective order to avoid extensive construction of test stubs.

Identification of changes and their impact is only one aspect of software maintenance. We are currently extending the capabilities to include various metrics, such as complexity, object-orientedness, and effort, to facilitate the maintenance work. The system is being expanded with these new fea-
tures and integrated with a test environment to provide retesting support.

8 Acknowledgment

The material presented in this paper is based on work supported by the Texas Advanced Technology Program (Grant No. 003656-097), Fujitsu Network Transmission Systems, Inc., Hewlett-Packard Company, and the Software Engineering Center for Telecommunications at UTA.

9 References

Specification-Based Approach,” submitted for publication.


