JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. 2000; **00**:1–2 *Prepared using* smrauth.cls *[Version: 2001/03/16 v1.01]*

Research

An Improved Method of Selecting Regression Tests for C++ Programs

Y. K. Jang[1*], M. Munro[2], and Y. R. Kwon[1]

[1] *Department of Electronic Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusong-gu, Taejon, 305-701, Korea*
[2] *Department of Computer Science, University of Durham, Science Laboratories, South Road, DURHAM, DH1 3LE, U.K.*

SUMMARY

This paper describes an impact analysis technique that identifies which parts should be retested after a system written in C++ is modified. We are interested in identifying the impacts of changes at the class member-level by using dependency relations between class members. We try to find out which member functions need unit-level retesting and which interactions between them need integration-level retesting. To get precise analysis results, we adopt a technique that classifies types of changes and analyze the impact for each type. Primitive changes, changes which are associated with C++ features, are first defined and their ripple effects are computed in order to construct a firewall for each type of changes systematically. We have applied our prototype tool to a real system with small size. This case study shows some evidence that our approach gives reasonable efficiency and precision as well as being practical for analyzing change impacts of C++ programs.

KEY WORDS: regression testing; change impact analysis; object-orientation; firewall; dependency relation

1. Introduction

Whenever a program is modified, it must be retested to ascertain whether changes have been made correctly and whether those changes have caused any adverse effect on its behavior. However, testing is a complicated and expensive activity: some studies have shown that more than 50% of development effort in the life cycle of a software program is spent on testing and when maintenance is included, nearly two thirds of the development effort[1, 2]. Therefore, several selective retesting techniques have been developed in order to reduce the time and effort of retesting.

Some essential issues in selective retesting techniques are: (1) how to identify changes of a program and the affected components, (2) how to maintain test suites during evolution of a program, (3) what test strategy should be used to retest these affected components, and (4) how to select reusable test cases and generate new ones if necessary[3]. Among these issues, we focus on change identification and impact analysis.

Identifying the impacts of changes must be reasonably precise so that we can isolate as many parts of the program as possible from retesting[4]. On the other hand, there is an observation that the more precise an approach is, the less it becomes efficient[5]. For example, an approach which identifies statements to be retested is more precise than an approach which chooses functions as a retesting unit, but the former can be less efficient than the latter. Moreover, it must be supported by automated tools because retesting is a time consuming activity which requires dependency information between components of a program. In

*Correspondence to: Y. K. Jang, Department of Electronic Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusong-gu, Taejon 305-701, Korea. E-mail: ykjang@salmosa.kaist.ac.kr

this paper, we aim to devise an approach which maintains a balance between precision and efficiency and which supports automation.

Object-oriented languages such as C++ and Java include concepts such as inheritance, polymorphism, and dynamic binding. These features not only result in more complex dependencies between program entities but also make dependency analysis more difficult[6]. We have previously developed an approach for analyzing change impact of a C++ program[7]. Although our approach was basically based on the class firewall method[8], but we are interested in identifying which member functions(of a class instead of a class on a while) had to be retested after modification. This paper systematizes our previous approach in identifying change impact of C++ programs in an effort to select as a small number of retesting sets as possible and discusses on a tool which implements our approach. We first define changes associated with C++ features as primitive changes and compute their impacts. Then, we classify changes that can occur in a C++ program and construct a member-level firewall for each type of classified changes by using the firewalls of primitive changes. We also describe the results of a case study using a small example program to show whether our approach provides reasonable precision and efficiency in assessing the impacts of changes. Our approach regards a C++ program as our target program because it is being widely used, but it can be made applicable to other object-oriented languages such as Java.

The rest of our paper is organized as follows. Section 2 presents previous researches related to regression testing and change impact analysis. Section 3 describes primitive changes and their firewalls. It also presents changes which may occur in a C++ program and how to construct their firewalls by using primitive cases. In Section 4, we explain the structure of our tool and how our tool works using a small example. Section 5 describes the results of a case study with a software system written in C++. Section 6 summarizes contributions of this work and suggests further research directions.

2. Related Works

Some approaches for analyzing change impact of object-oriented programs have been developed. Rothermel and Harrold[9] used dependence graphs that represent both control dependency and data dependency at the statement-level in an abstracted form of classes and application programs. They constructed dependence graphs for both the original program and the modified program, and then observed the differences between the two graphs by comparing corresponding nodes during the graph-traversal. Although the results of impact analysis might be very precise, data flow analysis is often restricted to the intra-procedural level and its computational complexity may be costly. Therefore, application of this approach seems to be restricted to programs with a small size. On the other hand, Kung et al.[8] introduced a notion of class firewall based on three dependency relations between classes - inheritance, association, and aggregation - to identify the effects of a class-level modification. This approach is less precise than that of Rothermel and Harrold[9], but more efficient for large software systems. The fundamental difference between these two approaches is the granularity of retesting units and dependency information used in analysis. While Rothermel and Harrold regarded a statement as a retesting unit and used the statement-level dependency information, Kung et al. considered a class a unit of retesting and used the class-level dependencies. In our approach, we regard a member function as a retesting unit and use dependency information at the member-level because it is expected to give a reasonable precision and efficiency in analyzing change impacts.

Some approaches have categorized the code changes that may be made to object-oriented software and analyzed how these changes affect other classes in the system in order to obtain reasonable precision and efficiency[10, 5, 11]. Kung et al.[10] provided a regression test model that consists of object relations and the interface, control structure of a member function in a class, and relationships to other data items and function members of classes. They dealt with data change, method change, class change, and class library change and described how to identify each type of change. However, their computed firewall was restricted to identifying the effect of a class change at the class-level; impact analysis for other change types is needed. Li and Offutt[5] first analyzed how encapsulation, inheritance, and polymorphism would affect the impacts of changes and described algorithms to identify potentially affected classes by using transitive closure dependency relations between class members. They noted that it is possible to optimize their algorithms by categorizing the possible code changes and by giving each change an attribute according to the degree of

its influence on other classes. Although this approach is more precise than that of Kung et al., there exists a room for reducing the impacts identification efforts because they use transitive closure dependency without considering types of changes; retesting the changed class is not always necessary and sometimes, changes may not be propagated to other parts which have transitive closure dependency[12, 11]. Furthermore, they did not consider changes to inheritance relations and virtual member functions. Our approach basically follows this approach, but we hope to identify change impacts more precisely by utilizing information on the types of changes.

Chaumun et al.[11] defined 63 changes which can be made in a class, a member function, and a variable and used association, aggregation, inheritance, and invocation links to analyze the impact with. Their experiment with a small C++ program showed that there was no impact for 22 of these changes, there was only local impact for 4 changes, and there was impact in other classes for 37 changes. Rangaraajan et al.[12] assumed that any change to a C++ class must be a permutation from a finite set of atomic changes, minimal units that the program compiles and links. As a criterion to judge whether a member function needs to be retested after a sequence of atomic changes, they used the information of all symbols this function binds to statically. They all found that some changes do not require retesting.

Vokolos and Frankl[13] have presented a textual differencing technique that compares source files from the old and the new versions of a program in written C. They used *diff*, the file comparison program, as a comparison tool in order to determine the differences in the program texts. They showed that their approach is adequately effective in practice through empirical results. On the other hand, Chen et al.[14] have developed a regression test selection tool in which *diff* identifies modifications to the code entities(functions, variables, types, and preprocessor macros) of a C program and their dependency relations. It is a relatively coarse-grained analysis, but produces a reasonable and practical tradeoff between granuality of analysis and time/space complexity. This research showed the practicality of impact analysis tools which are implemented using *diff*.

From the brief survey above, we observe that many of the previous approaches are often biased either by efficiency or precision and the change impact models are often incomplete or not systematic. We also note that retesting efforts can further be reduced by analyzing more precisely the impact of changes.

3. Change Impact Analysis

In this section, we present an approach to identifying code changes and their impacts automatically. We are interested in identifying member functions that should be retested when a C++ program is modified. Also, we aim at constructing a change impact model as systematically and precisely as possible for each type of change. First, we classify types of changes and analyze the impact for each type. We define the impacts of the changes associated with C++ features to compute a firewall for each type of changes.

3.1. Categorization of Changes

In order to identify member functions affected by changes, dependency relations such as invocation relation between member functions, data definition/use relation between a member function and a data member, and inheritance relation between member functions are used. Invocation means that a member function is called by another member function. A data definition/use dependency is established when a member function(or a global function) defines/uses the value of a data member. We call these functions definition/use functions of the data member. In our approach, we assume that the value of a data member is used or defined only by member functions or global functions. Regarding inheritance relation as an incremental modification, a class member is classified into *new*, *recursive*, and *redefined* according as to whether it is inherited from the parent class or not[15].

We consider levels of changes according as where these changes are made. Four levels of changes in a C++ program are considered: changes at the level of a data member, a member function, a class, and the inheritance relation. We explain each type of change in more details below.

- **Types of change at the level of a member function:** Changes to a member function can be made in three different ways: addition or deletion of a member function, changes to its interface, and changes

to its implementation. Therefore, addition of a new member function, deletion of a member function, changes of virtuality, visibility, signature, and a change to the implementation are regarded as types of change at this level. As a change to the implementation, we take into consideration changes to data definition/use relation and invocation relation because we use the member-level dependency relations.

- **Types of change at the level of a data member:** For a data member, its visibility or data type can be changed and a new data member may be added, or an existing data member may be deleted. Changing the value of a data member is considered a change to the implementation of its definition member function that modifies the value of the data; therefore, this type of change is not a change type at this level.
- **Types of change at the level of a class:** Changes at the class-level are addition of declaration of a new class and deletion of declaration of an existing class. Note that changes are made at different levels. For example, if we want to add a new class A with a member function m, two changes must be made; first, A is added, which is a change at the class-level and then, m is added to A, which is a change at the member function-level.
- **Types of change at the inheritance relation:** These changes include adding or deleting an inheritance relation between two existing classes. When a new inheritance relation is added, the implementation of member functions in the derived class can be modified in order to use newly inherited members from the base class. On the other hand, if an inheritance is deleted and the member functions in the derived class can no longer use inherited members, their implementation must be changed.

Table I lists a total of 30 types of changes. We are not convinced whether our classification about changes is complete or not, but included all changes that can be generally made. Basically, when one type of change is made to a system and impact analysis for that change is performed. However, we also allow restricted multiple types of change at the same time. As an example, when a member function $m1$ is added(addition of a member function), it can invoke existing member functions(change to implementation of $m1$) and another member function $m2$ can be modified in order to invoke the added function(change to implementation of $m2$). We define these multiple types of change as **changes in the same change scope**. Table I also shows changes in the same change scope as each type of change. Whenever a program undergoes several changes in the same change scope, we calculate change impacts using the algorithms to be discussed in Section 3.3.

[Table 1 about here.]

3.2. Impact Analysis for Primitive Changes

In software systems written in a procedural language, dependencies between functions or data tend to be explicit. However, a C++ program includes features that might cause implicit dependencies. Access control to members encapsulated in other classes creates more complex dependencies along with inheritance relations which allow members in a derived class to use members defined in its parent classes. Moreover, dynamic binding makes invocation relations between member functions unclear because it allows the decision on implementation to be delayed until run-time. We call changes associated with these C++ features **primitive changes** and define the following eight primitive changes.

- **Changes to the scope of a member function:** The scope of a member function is defined as member functions which it invokes and as data members whose values are defined and used. When a member is added or deleted, the scope of other member functions which interact with it can be changed implicitly as well as explicitly. Cases of addition and deletion of a member function are defined as Cases 1 and 2, respectively.
- **Changes related to dynamic binding:** When a virtual member function is called, it is difficult to find out which function is actually being invoked among it and its virtual-redefined functions. Whenever there exist implicit dependencies from their calling functions to virtual functions, those dependencies should be retested. We define four more primitive cases.
  - Case 3: Invocation to a virtual member function is added.

  - Case 4: An implicit invocation can be raised by an existing dependency when a virtual-redefined member function is added to a derived class.
  - Case 5: When the virtuality of a member function in a base class is changed(a virtual member function is changed into a non-virtual function or vice versa), an implicit invocation with a redefined member function in its derived class can be created.
  - Case 6: A new invocation can be created because of the change of a member function's interface in a base or a derived class.

- **Changes to data definition/use:** Case 7 is a change to definition of a data member in its definition member function and Case 8 is a change to use of a data member in its use member function. To assess the impacts of these changes, we use data definition/use dependencies that are shown at the member-level. Such information is less precise than data dependency at the statement-level, but it helps to avoid the complexity of analyzing data-flow at the statement-level.

The impacts of changes in the same change scope which were defined in Section 3.1 is composed of those primitive changes. First, we find out the impacts of primitive changes in order to identify a member-level firewall for a type of change. Figures 1 and 2 summarize the impacts for each of primitive change. A firewall(F) is one of three types: a unit firewall(F_U), a set of member functions which require unit-level retesting, an integration firewall(F_I), a set of interactions between member functions which require integration-level retesting, and a definition-conflict check firewall(F_{DEF}), a set of member functions which need to check whether their definitions for it are conflicting, when they define the value of a given data member.

We briefly explain the impacts of some primary cases. Figure 1(b) shows an example of Case 1: an invocation from $Derived.m2$ to $Base.m3$ is changed to the invocation from $Derived.m2$ to $Derived.m3$ when a new member function $m3$ is added to a class $Derived$. Then, the functionality of $Derived.m2$ is changed because it invokes $Derived.m3$ and it might have an influence on other member functions which invoke it. Therefore, both the member function added and other member functions that call it directly or indirectly need retesting at the unit-level and the integration-level.

Figure 1(d) describes Case 3 in which $Base.m1$ has a new implicit interaction with $Derived.m3$ when the program is changed such that an invocation from $Base.m1$ to a virtual member function $Base.m3$ is added. In this case, $Base.m1$ and member functions which invoke it are included in F_u because $Base.m1$ might have a different functionality. Also, the interactions between them and newly created interactions require integration-level retesting.

Figure 2(c) shows an example of Case 7 in which a definition member function $Base.m1$ of a data member $d1$ modifies the value of $d1$. Then, the definition member functions of $d1$ must be tested at the integration-level in order to check whether new definition for the value of $d1$ is in conflicts with other definitions or not[16]. Also, we must test the usage of new definition to confirm that this change is correct. As a way of identifying the impacts of definition/use of data, we use **a changed definition-use pair**(when $m1$ changes the value of $d1$ or $m2$ modifies the use of $d1$, a pair of $m1$ and $m2$ is a changed definition-use pair). We assume that the usage of the changed definition is properly tested if there is no error between the changed definition and any use of the data member. It is less precise than data flow analysis at the statement-level, but it is more efficient, and still gives a reasonable precision. We do not explain other primitive cases shown in Figures 1 and 2 because they are similar to cases described before. More detailed explanation can be found in [7].

[Figure 1 about here.]

[Figure 2 about here.]

3.3. Impact Analysis for Each Type of Changes

The firewall for each type of changes is calculated by using the firewalls of primitive changes developed earlier. The complete algorithms for constructing them have been explained previously [7]. For purposes of illustration, we describe how to construct a firewall when a member function is added:

Addition of a member function: If an added member function defines(uses) the value of a data member, it is supposed that the added function has changed the definition(use) of the data. In this case, a firewall is constructed by combining the firewall of Case 7(Case 8) and itself. $F_U(m,\ d,\ case\ 7)$, $F_I(m,\ d,\ case\ 7)$, and $F_{DEF}(m,\ d,\ case\ 7)$ indicate F_U, F_I, and F_{DEF} of the Case 7 respectively, where m is a modified definition member function of a data member d. On the other hand, if a redefined or virtual-redefined new function is added to a derived class, new dependencies such as the Cases 1 or 4 can be created. Then, the firewall is combined with the firewalls, $F_U(m,\ case1(4))$ and $F_I((m,\ case1(4))$, where m is the added member function.

When an added function uses virtual member functions, it is regarded as a function which has been modified in order to invoke virtual functions like Case 3. Thus, its firewall is used to identify its impacts. Also, some member functions can be modified in order to invoke the added function. In this case, member functions which invoke the added one directly or indirectly need unit-level retesting and their interactions need integration-level retesting. The algorithm constructing firewall for addition of a member function is summarized as follows:

1. Let an added member function be m_{add}. Then, $F_U = F_U \cup \{m_{add}\}$
2. If m_{add} is a definition member function of a data member d,
   (a) $F_U = F_U \cup F_U(m_{add},\ d,\ case\ 7)$
   (b) $F_I = F_I \cup F_I(m_{add},\ d,\ case\ 7)$
   (c) $F_{DEF} = F_{DEF} \cup F_{DEF}(m_{add},\ d,\ case\ 7)$
3. If m_{add} is a use member function of a data member d,
   (a) $F_U = F_U \cup F_U(m_{add},\ d,\ case\ 8)$
   (b) $F_I = F_I \cup F_I(m_{add},\ d,\ case\ 8)$
   (c) $F_{DEF} = F_{DEF} \cup F_{DEF}(m_{add},\ d,\ case\ 8)$
4. If m_{add} is a redefined or virtual-redefined member function,
   (a) $F_U = F_U \cup F_U(m_{add},\ case\ 1)$
   (b) $F_I = F_I \cup F_I(m_{add},\ case\ 1)$
5. If m_{add} is a virtual-redefined member function,
   (a) $F_U = F_U \cup F_U(m_{add},\ case\ 4)$
   (b) $F_I = F_I \cup F_I(m_{add},\ case\ 4)$
6. If m_{add} invokes a non-virtual member function m_j,
   (a) $F_I = F_I \cup interaction(\{m_j\},\ \{m_{add}\})$
7. If m_{add} invokes a virtual member function m_j,
   (a) $F_U = F_U \cup F_U(m_{add},\ case\ 3)$
   (b) $F_I = F_I \cup F_I(m_{add},\ case\ 3)$
8. If m_j invokes m_{add},
   (a) $F_U = \cup_{i=0}^{n}\ call^i(m_j)$
   (b) $F_I = \cup_{i=0}^{n-1}\ interaction(call^i(m_j),\ call^{i+1}(m_j))$

4. Prototype Implementation

We have developed a prototype system which implements our approach and conducted a case study to show its practicality. The results of our case study will be discussed in Section 5. First, we describe the configuration of our prototype system and explain how the system works with a simple example.

4.1. Tool Architecture

Figure 3 shows the overall structure of our prototype system. This prototype system has been developed by integrating the following programs: *cpp*, the C/C++ preprocessor; *diff*, the general purpose file comparison program; *gen++*, an analyzer generator and a tool generation facility for C++[17], and *GraphTool*, a graph layout tool developed in RISE at the University of Durham(http://www.dur.ac.uk/RISE). This system automatically identifies the type of change and member functions potentially affected by the change. It consists of three components: a program analyzer, an impact analyzer, and a graph tool.

[Figure 3 about here.]

Program Analyzer: Our program analyzer was generated from the query program developed using *gen++*. It takes a C++ program preprocessed by *cpp* as input, and outputs dependency information on the input program for impact analysis. One of the output files is *.2dg* in a textual graph format for *GraphTool*. Another output file, *.cmp*, has information about entities(classes, data members, and function members), inheritance relations, and member-level dependencies of the input program.

Impact Analyzer: After the old and the new versions of a program are processed by the program analyzer, *diff* compares their *.cmp* files which contain dependency information. While Vokolos and Frankl[13] made the source-to-source comparison with *diff*, we compared dependency information of two programs in order to determine the impacts more precisely. Although the difference in the program texts has no impacts on dependence relation, test cases which cover this difference are selected as reusable ones. The impact analyzer examines the output of *diff* to determines which type of change is made in the old version of the program. We will briefly explain how to determine the type of change in Section 4.2. It subsequently constructs a member-level firewall according to the firewall construction algorithm of the identified type of change described in Section 3.3.

GraphTool: It helps us understand the structure of the subject program and dependency relations between program entities by presenting it in a graphical form. It also compares the graphs of old and new versions of the input program.

4.2. An Example

[Figure 4 about here.]

Figure 4 shows an example program to be used to explain how our system works. Figures 4(a) and 4(b) are the old and the new versions of the program, respectively. The rectangular portion in Figure 4(b) shows which parts of the program were changed in the old version. In this example, the type of change is addition of a new member function; a member function $x3$ is added to the class Y and $Y :: y4$ is changed to invoke it. The program analyzer generates output files, *ex1.cmp, ex1.2dg, ex2.cmp*, and *ex2.2dg*, which include dependency information of $ex1$ and $ex2$. These files are in the textual graph format for *GraphTool*, which is composed of nodes that represent class, class member, global data/function, and function parameter and edges that indicate dependencies between nodes. Table II(a) shows a template of a node which represents a member function; <*+Node*> and <*-Node*> indicate the beginning and the end of definition of a node, $[m]$ means that this node represents a member function, $[visibility]$, $[virtuality]$, and $[type]$ has the interface information about the member function, *qualifier::name* represents the name of the member function and its qualifier, and a *defined class* maintains a name of a class which defines the member function. Similarly, Tables II(b)∼(e) show a template of a node which represents a data member and edges which indicate member-level dependencies.

[Table 2 about here.]

[Table 3 about here.]

[Table 4 about here.]

In this example, *diff* creates a file, *ex1_ex2.diff*, which includes textual differencies between *ex1.cmp* and *ex2.cmp*. Since the *.diff* file has a unique format according to the type of change, the impact analyzer can identify which type of change was made in the program. Table III shows a pattern of a *.diff* file when a new member function is added to the program. Table III(a) indicates that a new member function is added; (b) and (c) show that the added function defines and uses the value of an existing data member, (d) and (e) show that the added function invokes a member function, (f) shows that a member function invokes the added one, and (g) shows that since the added function is redefined or virtual-redefined function, primary changes such as Case 1 and Case 4 happen.

The impact analyzer first identifies the type of change; it recognizes that a new member function was added to the new version when the format of Table III(a) is recognized, (b)∼(f) are changes in the same change scope as (a), and (g) was made additionally because of (a). Moreover, *.diff* includes information on other changes that were affected by the above changes. Table IV shows the *ex1_ex2.diff* file of this example. The impact analyzer interprets each part of this output file as follows:

(a) A new member function $Y :: x3$ is added.
(b) $Y :: x2$ invokes $Y :: x3$.
(c) $Y :: x3$ defines the value of $Y :: d3$.
(d) $Y :: x3$ uses the value of $X :: d1$.
(e) $Y :: x3$ invokes $Y :: y1$.
(f) $Y :: y4$ invokes $Y :: x3$.

Upon determining a type of change, the impact analyzer computes its firewall using the firewall construction algorithm corresponding to that type. As shown in Table III, (a)∼(f) are matched with appropriate parts of the algorithm explained in Section 3.3. Table V shows the computed firewall of this example. For each of (a)∼(f), unit firewall and integration firewall were computed; unit firewall includes member functions(e.g., $Y :: x3$) to be retested at the unit-level, and integration firewall includes interactions(e.g., $X :: x4 - Y :: y3$) between member functions to be retested at the integration-level and interactions(e.g., $Y :: x1 - Y :: x3$) to be retested in order to check whether they are in conflicts with the definition of a data member or not. In this paper, we do not address how to retest member functions and interactions in the computed firewall.

[Table 5 about here.]

5. A Case Study

We have applied this prototype tool to a small target program in order to show its practicality. We hoped to explore in particular, (1) whether it can identify types of change from the difference information obtained by comparing the textual dependency of two versions of a program, and (2) whether our approach, which analyzes the impacts according to the type of change, is applicable to a realistic software system.

The software used in our experiment is a drawing program which consists of 11 program files and 18 header files. This program defines 26 classes to support various drawing functions[18]. Without any other version of this software available, we had to make new versions of this software by hand. For every class, we made a new version by deleting each member function except its constructor or destructor at a time. Thus, as many second versions as the member functions were created. Each version was modified to be compiled without any error: declaration of a member function, its implementation, and the invocation from other member functions to the deleted one were removed. We then analyzed the original version and each new version to determine the impacts. Since the $Application$ class included in this software has seven member functions, seven second versions were prepared and seven firewalls were obtained as the result of analysis for each version. We combined these results to produce one firewall because we were simply interested in identifying how many member functions are affected by this type of change - deletion of a member function defined in a class.

Table VI shows all the classes defined in this software and the number of member functions defined in each class. The results of impact analysis for deletion of a member function consist of a unit firewall, an integration firewall and a set of member functions for checking definition conflicts. Classes in a drawing program can be classified into four groups as follows:

- **The firewall includes no class.** For 10 classes($DOSKeyboard$, $EllipseTool$, $EventHandler$, $GraphicsScreen$, $Line$, $LineTool$, $Node$, $Poller$, $Rectangle$, and $RectangleTool$), deletion of a member function had no impact on any other classes. We did not need to consider this type of change for class $Node$ because it has no member function. Even though other classes defined some member functions which have their implementation, this change did not affect other parts of the software because no member function invokes them nor they define or use data members.

- **The firewall includes only the changed class.** The class $DiagramEditor$ had an influence only on itself. It has 8 public member functions: 2 member functions are without implementation, and 6 member functions which are used only within the class $DiagramEditor$.
- **The firewall includes classes other than.** Six classes($Application$, $Display$, $Keyboard$, $Mouse$, $Tool$, and $ToolManager$) had influence on other classes, but not on themselves. Although classes $Application$, $Keyboard$, and $Tool$ have 4 member functions in their firewall, they do not affect other classes because they include only pure virtual member functions.
- **The firewall includes both the changed class and other classes.** Nine classes($Collection$, $CollectionIterator$, $ControlPoint$, $CreationTool$, $Drawing$, $Element$, $Ellipse$, $MSMouse$, and $SelectionTool$) affected other classes as well as themselves.

[Table 6 about here.]

Table VI lists the number of classes included in the firewall obtained by our approach and in the firewall obtained by the class firewall approach[3] separately. As expected, in most of classes, the size of our firewall turned out to be smaller than that of the class firewall approach. This is understandable because isolating their impacts according to the types of changes is more precise than the class firewall approach which includes the impact of all types of changes. However, in the cases in which a member function was deleted in $CollectionIterator$, $Ellipse$, $MSMouse$, and $SelectionTool$ classes, our approach resulted in larger firewalls than the class firewall approach. The reason appeared to be due to association relations between classes: our approach detected more precise association relations by analyzing dependency relations among class members.

This case study shows that our approach works for a realistic software system; it has demonstrated that it is possible to identify a type of change from the difference information obtained by comparing textual dependencies at the member-level of two programs and to analyze the impacts of that. A case study on only one type of change, deletion of a member function, was conducted because creating several versions of the target programs manually was time consuming. Therefore, it is desirable to have more empirical results for other types of changes to have more evidence that our approach produces reasonable efficiency and precision.

6. Conclusions

In this paper, we described an approach which constructs firewalls at the class member-level as a mechanism for identifying the impact of a change to a software system written in C++. Techniques for analyzing dependencies among program statements may provide accurate impact results, but they are applicable only for programs of a small size because of a high cost. On the other hand, methods of analyzing at the class-level tend to select too many parts as the results of the impacts analysis even when changes are very small in a program. In our approach, a member function was considered a unit of impact identification to attain reasonable precision and efficiency. We attempted to reduce retesting efforts by classifying the types of change and identifying the impacts of each type. Although the classification of type of change may not be complete, our classification scheme is being widely used[5, 10, 11] and we believe that it represents the most common changes which are made in a C++ program.

We have implemented a prototype system which implements our approach by integrating *cpp*, *diff*, *gen++*, and *GraphTool*. Our experiment with a small drawing program showed that our approach which uses classification of changes was able to identify the type of change using the difference information obtained by comparing dependency information of texts of the two programs, and analyze impacts of the change. We showed that our approach identified change impacts more precisely that a class-level approach did. Further experiments with larger systems are needed to show that our approach produces reasonable efficiency and precision in selecting a set of retesting test cases. Comparison of our approach with other techniques such as [5, 11, 13, 9] are also desired. This paper has focused on the activity of change impact analysis as a starting point of a research on regression testing. We need to consider other issues of regression testing such as test suite maintenance, test strategy, and new test generation.

REFERENCES

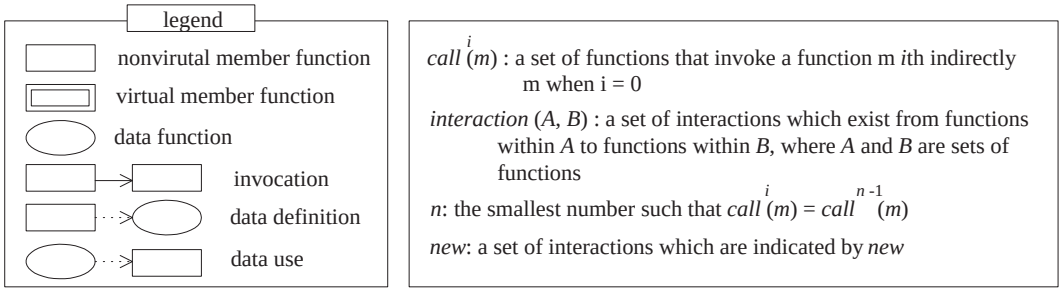
1. E. Kit. *Software Testing in the Real World: Improving the Process.*(1st edn). Addison-Wesley, 1995.
   B. P. Lientz, E. B. Swanson. *Software Maintenance Management.*(1st edn). Addison-Wesley, Massachusetts, 1980.
2. G. J. Myers. *The Art of Software Testing*(1st edn). Wiley-Interscience, New York, 1979.
3. P. Hsia, X. Li, D. C. Kung, C. T. Hsu, L. Li, Y. Toyoshima, C. Chen. A technique for the selective revalidation of OO software. *J. Software Maintenance: Research and Practice* Jul/Aug 1997; **9**(4):217–233.
4. G. Rothermel, M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Software Engineering* Aug 1996; **22**(8):529–551.
5. L. Li, A. J. Offutt. Algorithmic analysis of the impact of changes to OO software. *Proc. IEEE Int'l Conf. Software Maintenance* 1996;171–184.
6. N. Wilde, R. Huitt. Maintenance support for OO programs. *IEEE Trans. Software Engineering* Dec 1992; **18**(12):1038–1044.
7. Y. K. Jang, H. S. Chae, Y. R. Kwon, D. H. Bae. Change impact analysis for a class hierarchy. *Proc. Asia-Pacific Software Engineering Conference* 1998;304–311.
8. D. C. Kung, J. Gao, P. Hsia. Class firewall, test order, and regression testing of OO programs. *J. Object-Oriented Programming* May 1995; **8**(2):51–65.
9. G. Rothermel, M. J. Harrold. Selecting regression tests for OO software. *Proc. IEEE Int'l Conf. Software Maintenance* 1994;14–25.
10. D. C. Kung, Gao, Jerry, Chen, Cris. On regression testing of OO programs. *J. Systems and Software* 1996; **32**(1):21–40.
11. M. A. Chaumun, H. Kabaili, R. K. Keller, F. Lustman. A change impact model for changeability assessment in OO software systems. *Proc. IEEE Third European Conf. Software Maintenance and Reengineering* 1999;130–138.
12. K. Rangaraajan, P. Eswar, T. Ashok. Retesting C++ classes. *Proc. Nineth Annual Software Quality Week* May 1996.
13. F. I. Vokolos, P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. *Proc. IEEE Int'l Conf. Software Maintenance* 1998;44–53.
14. Y. F. Chen, D. S. Rosenblum, K. P. Vo. TestTube: a system for selective regression testing. *Proc. Int'l Conf. Software Engineering* 1994;211–220.
15. M. J. Harrold, J. D. McGregor. Incremental testing of OO class structures. *Proc. Int'l Conf. Software Engineering* 1992;68–80.
16. D. E. Perry, G. E. Kaiser. Adequate testing and OO programming. *J. Object-Oriented Programming* Jan/Feb 1990; **2**(5):13–19.
17. P. T. Devanbu. GENOA - a customizable, language and front-end independent code analyzer. *Proc. Int'l Conf. Software Engineering* 1992;307–317.
18. M. Priestley. *Practical Object-Oriented Design.*(1st edn). McGraw-Hill Book Co., New York, 1996.

Yoon Kyu Jang is a Ph.D. student of Software Engineering in the Department of Electronic Engineering and Computer Science at the Korea Advanced Institute of Science and Technology(KAIST) in Korea. She received her B.S. degree in computer science from the KAIST, Korea, in 1996 and M.S. degree in computer science from the KAIST, Korea, in 1998. Her research interests include regression testing, software maintenance, and object-oriented testing.

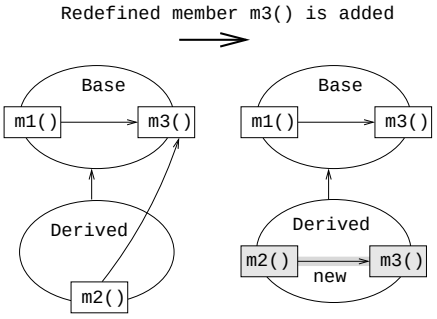
Malcolm Munro is a Professor of Software Engineering in the Department of Computer Science at the University of Durham in the UK. He has been an active researcher in software maintenance since 1987. His current research interests are in program comprehension, software visualization, reverse engineering, and reuse-reengineering. He has served on the program and organizing committees for international conferences. Malcolm is a founder member of the Centre for Software Maintenance and in the Research Institute for Software Evolution, both located at the University of Durham. His Ph.D. from the University of Durham is in Computer Science.

Yong Rae Kwon is a Professor of Software Engineering in the Department of Electronic Engineering and Computer Science at the Korea Advanced Institute of Science and Technology(KAIST) in Korea. He received B.S. and M.S. degrees in physics from Seoul National University, Korea, in 1969 and 1971 respectively, and a Ph.D. in physics from the University of Pittsburgh in 1978. He taught as an instructor at Korea Military Academy from 1971 to 1974. He was on the technical staff of Computer Science Corporation from 1978 through 1983 working on the ground support software systems for NASAs satellite rojects. He joined the Faculty of Computer Science of KAIST in 1983. His research interests include verification of real-time parallel software, object-oriented technology for real-time systems and quality assurance for highly dependable software.

List of Figures



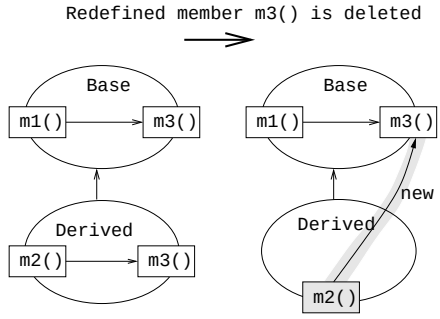
(a) Notations used in this figure



$F_U = \mathbf{U}_{i=0}^{n}$ call^{i}(added member function m)

$F_I = \mathbf{U}_{i=0}^{n-1}$ interaction(call^{i}(m), call^{i+1}(m))

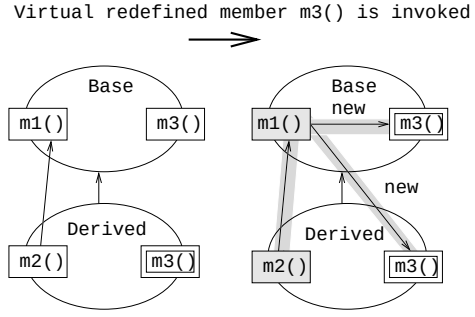
(b) Case 1: scope change due to addition of a member function



$F_U = \mathbf{U}_{i=1}^{n}$ call^{i}(deleted member function m)

$F_I = \mathbf{U}_{i=1}^{n-1}$ interaction(call^{i}(m), call^{i+1}(m)) $\mathbf{U}$ new

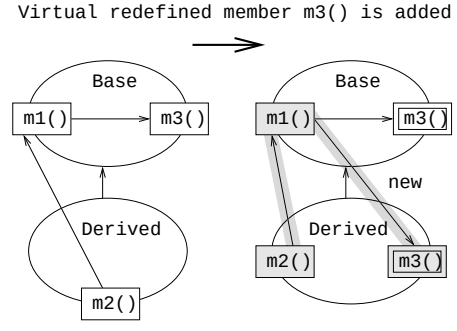
(c) Case 2: scope change due to deletion of a member function



$F_U = \mathbf{U}_{i=0}^{n}$ call^{i}(changed member function m)

$F_I = \mathbf{U}_{i=0}^{n-1}$ interaction(call^{i}(m), call^{i+1}(m)) $\mathbf{U}$ new

(d) Case 3: new interaction due to addition of a virtual call

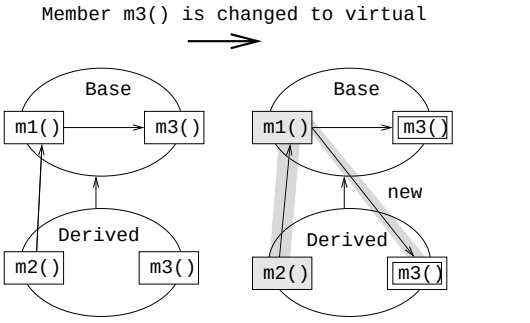


$F_U = \mathbf{U}_{i=0}^{n}$ call^{i}(added member function m)

$F_I = \mathbf{U}_{i=0}^{n-1}$ interaction(call^{i}(m), call^{i+1}(m))

(e) Case 4: new interaction due to addition of a virtual member

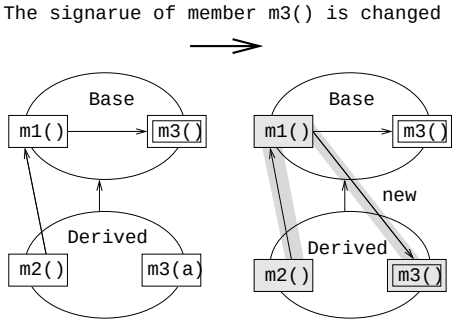
Figure 1. Primary Cases 1∼4 and their impacts



$F_U = \mathbf{U}_{i=1}^{n}$ call^i(changed member function m)

$F_I = \mathbf{U}_{i=1}^{n-1}$ interaction(call^i(m), call^{i+1}(m)) **U** new

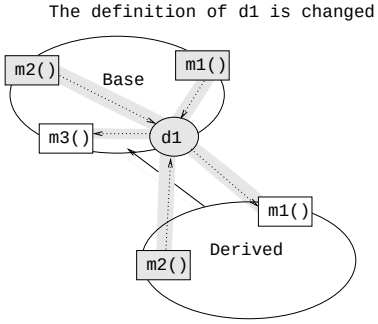
(a) Case 5: new interaction due to change to virtual



$F_U = \mathbf{U}_{i=0}^{n}$ call^i(changed member function m)

$F_I = \mathbf{U}_{i=0}^{n-1}$ interaction(call^i(m), call^{i+1}(m))

(b) Case 6: creation of a virtual call due to interface change

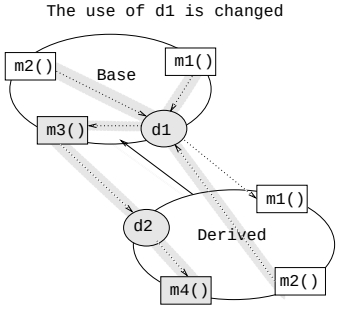


$F_U = \mathbf{U}_{i=0}^{n}$ call^i(changed definition function m)

$F_I = \mathbf{U}_{i=0}^{n-1}$ interaction(call^i(m), call^{i+1}(m))
U changed definition-use pairs

F_{DEF} = a set of definition functions of the data

(c) Case 7: change to data definition



$F_U = \mathbf{U}_{i=0}^{n}$ call^i(changed use function m)

$F_I = \mathbf{U}_{i=0}^{n-1}$ interaction(call^i(m), call^{i+1}(m))
U changed definition-use pairs

F_{DEF} = a set of definition functions of the data

F = F **U** F(m, data member defined by m, case 7)

(d) Case 8: change to data use

Figure 2. Primary Cases 5∼8 and their impacts

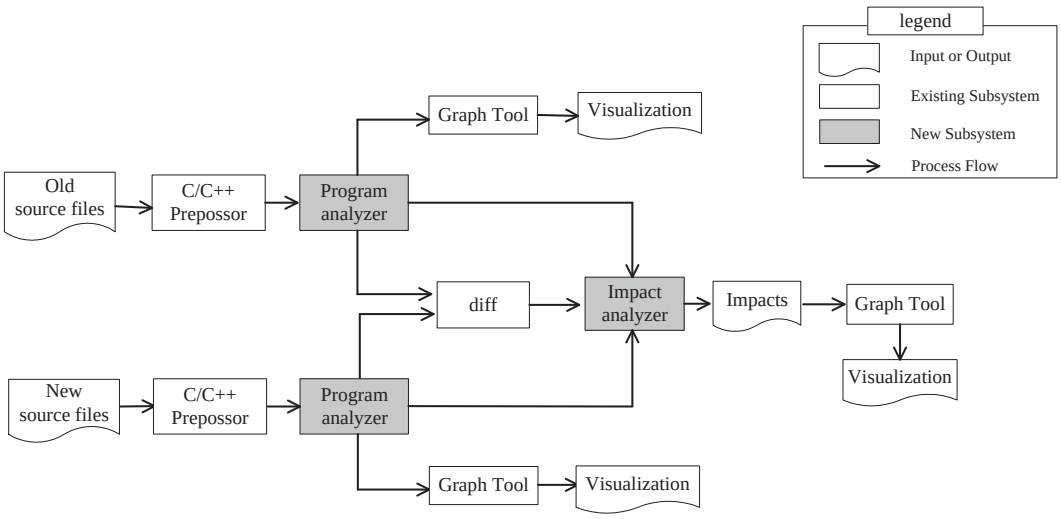


Figure 3. A Prototype implementation of impact analysis

```
class X {
        protected :
                int d1;
                float d2;
        public :
                void x1();
                int x2();
                virtual void x3();
                void x4();
};

class Y : public X {
        private:
                int d3;
        public:
                void x1();
                int x2();
                virtual void y1();
                void y2();
                void y3();
                int y4();
};

void X::x1() { d1 = 5; x2(); }
int X::x2() { x4(); return d2; }
void X::x3() { d2 = d1 + 10; }
void X::x4() { x3(); }
void Y::x1() { d1 = 10; d3 = 11; }
int Y::x2() { x3(); y2(); }
void Y::y2() { y1(); }
void Y::y3() { x1(); y4(); }
int Y::y4() { return 0; }
```

(a) The old version of the program: ex1

```
class X {
        protected :
                int d1;
                float d2;
        public :
                void x1();
                int x2();
                virtual void x3();
                void x4();
};

class Y : public X {
        private:
                int d3;
        public:
                void x1();
                int x2();
                void x3();
                virtual void y1();
                void y2();
                void y3();
                int y4();
};

void X::x1() { d1 = 5; x2(); }
int X::x2() { x4(); return d2; }
void X::x3() { d2 = d1 + 10; }
void X::x4() { x3(); }
void Y::x1() { d1 = 10; d3 = 11; }
int Y::x2() { x3(); y2(); }
void Y::y2() { y1(); }
void Y::y3() { x1(); y4(); }
void Y::x3() { d3 = d1; y1(); }
int Y::y4() { x3(); return 0; }
```

(b) The new version of the program: ex2

Figure 4. The example programs

List of Tables

Table I. Types of change and changes in same change scope

Changes at the level of a member function	Changes in same change scope
Addition of a new member function	When a member function is added, it can invoke existing member functions or define/use the value of data members. Also, other member functions can be modified in order to invoke the added function at the same time.
Deletion of a member function	When a member function is deleted, its implementation is deleted and the part of implementation of other member functions which have invoked the deleted function are deleted at the same time.
Change to virtuality:	
Virtual to non-virtual	If virtual-redefined member functions were declared in the derived classes of the class in which the changed member function is declared, the "virtual" keywords of these functions are deleted.
Non-virtual to virtual	If redefined member function were declared in the derived classes of the class in which the changed member function is declared, the "virtual" keywords can be added to these functions.
Change to visibility:	
Public to protected/private, protected to private	Member functions, which have invoked the changed function but do not have access authority to that any more, are modified in order to remove the invocation.
Private to protected/public, protected to public	Other member functions can be modified in order to invoke the changed function if they have access authority to that.
Change to signature	The implementation of the member functions, which have invoked the changed function, must be changed in order to fit the signature of the changed function.
Change to implementation: Addition/deletion of invocation	There is no other changes in same change scope.
Addition/deletion of definition/use to the value of a data member	There is no other changes in same change scope.

Changes at the level of a a data member	Changes in same change scope
Change to type	The definition/use member functions are changed to be compatible with the data.
Change to visibility: Public to protected/private, protected to private	Member functions, which have defined/used the changed data and do not have access authority to that any more, are modified in order to remove their usage to the data.
Private to protected/public, protected to public	Member functions can be modified in order to define/use the value of the changed data if they have access authority to that.
Addition of a new data member	Member functions can be modified in order to define/use the value of the added data.
Deletion of a data member	Member functions that have defined/used the value of the deleted data must be changed in order to remove their use of the data.

Changes at the level of a class	Changes in same change scope
Addition of a new class	Since the addition of a class and the addition of its members are considered different types of changes, these two changes cannot happen at the same time. Also, any relation between this class and other classes is not added at the same time.
Deletion of a class	Since the deletion of a class and the deletion of its members are considered different types of changes, these two changes cannot happen at the same time. This class must not have any relation with other classes before being removed.

Changes at the inheritance relation	Changes in same change scope
Addition of an inheritance relation	When a new inheritance relation is added, the implementation of member functions in the derived class can be modified in order to use newly inherited members from the base class.
Deletion of an inheritance relation	Since member functions declared in derived classes cannot use any member declared in the base class, their corresponding implementation is changed.

Table II. Examples of nodes and edges in *.2dg* and *.cmp* files

(a) **member function** node:
⟨+Node⟩
[m][visibility][virtuality][type]qualifier::name[defined class]
⟨-Node⟩
(b) **data member** node:
⟨+Node⟩
[d][visibility][type]qualifier::name[defined class]
⟨-Node⟩
(c) **invocation relation** edge ($m1$ invokes $m2$):
⟨+Edge⟩
"member function $m1$ node" "member function $m2$ node"
⟨Name⟩ "c"
⟨-Edge⟩
(d) **define**/**use relation** edge ($m1$ defines or uses $d1$):
⟨+Edge⟩
"member function $m1$ node" "data member $d1$ node"
⟨Name⟩ "d" or "u"
⟨-Edge⟩
(e) **inheritance relation** edge ($c1$ inherits $c2$):
⟨+Edge⟩
"class $c1$ node" "class $c2$ node"
⟨Name⟩ "I"
⟨-Edge⟩

Table III. Patterns for addition of a new member function in the *.diff* file

Patterns in the *.diff* file	firewall construction algorithm
(A) ... a ...	
(a) ⟩ ⟨+Node⟩	• firewall algorithm 1
⟩ ⟨Name⟩ ”[m] .....”	• If it is virtual redefined,
⟩ ⟨-Node⟩	Case 1: firewall algorithm 2.(d)
(b) ⟩ ⟨+Edge⟩ ADDED FUNCTION ”[d] .....”	
⟩ ⟨Directed⟩	
⟩ ⟨Name⟩ ”d”	
⟩ ⟨-Edge⟩	• Case 7: firewall algorithm 2.(a)
(c) ⟩ ⟨+Edge⟩ ADDED FUNCTION ”[d] .....”	
⟩ ⟨Directed⟩	
⟩ ⟨Name⟩ ”u”	
⟩ ⟨-Edge⟩	• Case 8: firewall algorithm 2.(b)
(d) ⟩ ⟨+Edge⟩ ADDED FUNCTION ”[m] ... [nonvirtual] ... ”	
⟩ ⟨Directed⟩	
⟩ ⟨Name⟩ ”c”	
⟩ ⟨-Edge⟩	• firewall algorithm 3.(a)
(e) ⟩ ⟨+Edge⟩ ADDED FUNCTION ”[m] ... [virtual] ... ”	
⟩ ⟨Directed⟩	
⟩ ⟨Name⟩ ”c”	
⟩ ⟨-Edge⟩	• firewall algorithm 3.(b)
(f) ⟩ ⟨+Edge⟩ ”[m] ..... ” ADDED FUNCTION	
⟩ ⟨Directed⟩	
⟩ ⟨Name⟩ ”c”	
⟩ ⟨-Edge⟩	• firewall algorithm 4
(B) ... c ...	
(g) ⟨ ⟨+Edge⟩ ”m1” ”m2, the redefined type of ADDED FUNCTION”	
⟨ ⟨+Edge⟩ ”m1” ADDED FUNCTION	• Case 4: firewall algorithm 2.(c)

Table IV. The comparison result of *ex1.cmp* and *ex2.cmp* files by using *diff*

```
(a) 15a16
    > ⟨Name⟩ "[m][public][virtual][void Y::( )]Y::x3[Y]"
    63a65,82
(b) > ⟨+Edge⟩ "[m][public][nonvirtual][int Y::( )]Y::y4[Y]" "[m][public][virtual][void Y::( )]Y::x3[Y]"
    > ⟨Directed⟩
    > ⟨Name⟩ "c"
(c) > ⟨+Edge⟩ "[m][public][virtual][void Y::( )]Y::x3[Y]" "[d][private][int ]Y::d3[Y]"
    > ⟨Directed⟩
    > ⟨Name⟩ "d"
(d) > ⟨+Edge⟩ "[m][public][virtual][void Y::( )]Y::x3[Y]" "[d][private][int ]Y::d3[Y]"
    > ⟨Directed⟩
    > ⟨Name⟩ "u"
(e) > ⟨+Edge⟩ "[m][public][virtual][void Y::( )]Y::x3[Y]" "[d][protected][int ]X::d1[X]"
    > ⟨Directed⟩
    > ⟨Name⟩ "u"
(f) > ⟨+Edge⟩ "[m][public][virtual][void Y::( )]Y::x3[Y]" "[m][public][nonvirtual][void Y::( )]Y::y2[Y]"
    > ⟨Directed⟩
    > ⟨Name⟩ "c"
(g) > ⟨+Edge⟩ "[m][public][virtual][void Y::( )]Y::x3[Y]" "[m][public][nonvirtual][void Y::( )]Y::y3[Y]"
    > ⟨Directed⟩
    > ⟨Name⟩ "c"
```

Table V. The firewall of the example program

Changes	Unit firewall	Integration firewall
(a) A new member function Y::x3 is added. It is a virtual redefined function.	Y::x3, X::x4, X::x2, X::x1	X::x4 - Y::y3, X::x2 - X::x4, X::x1 - X::x2
(b) Y::x2 invokes Y::x3.	Y::x2	Y::x2 - Y::x3
(c) Y::x3 defines the value of Y::d3.	Y::x3, Y::x2, Y::y4, Y::y3	Y::x2 - Y::x3, Y::y4 - Y::x3, Y::y3 - Y::y4 definition conflict: Y::x1 - Y::x3
(d) Y::x3 uses the value of Y::d1.	Y::x3, Y::x2, Y::y4, Y::y3	Y::x2 - Y::x3, Y::y4 - Y::x3, Y::y3 - Y::y4, X::x1 - Y::x3, Y::x1 - Y::x3 definition conflict: Y::x1 - Y::x3
(e) Y::x3 invokes Y::y1.	-	Y::x3 - Y::y1
(f) Y::y4 invokes Y::x3.	Y::y4, Y::y3	Y::y3 - Y::y4

Table VI. The firewall for deletion of a member function

classes	the number of member functions				the number of classes	
	in this class	in unit firewall	in integration firewall	to check definition conflicts	in the firewall from our approach	in the firewall from class firewall approach
Application	7	2	0	0	1	4
Collection	2	5	3	0	4	10
CollectionIterator	2	5	5	0	3	1
ControlPoint	6	20	22	0	6	16
CreationTool	7	4	2	2	2	5
DiagramEditor	8	3	0	0	1	1
Display	9	30	23	0	11	21
DOSKeyboard	2	0	0	0	0	1
Drawing	2	3	0	0	3	8
Element	11	20	20	1	6	15
Ellipse	3	11	11	0	5	3
EllipseTool	2	0	0	0	0	2
EventHandler	1	0	0	0	0	2
GraphicsScreen	9	0	0	0	0	1
Keyboard	2	1	0	0	1	3
Line	2	0	0	0	0	3
LineTool	2	0	0	0	0	2
Mouse	15	9	0	0	2	4
MSMouse	15	10	15	8	2	1
Node	0	0	0	0	0	11
Poller	1	0	0	0	0	2
Rectangle	1	0	0	0	0	3
RectangleTool	2	0	0	0	0	2
SelectionTool	4	3	0	6	3	2
Tool	4	4	0	0	1	7
ToolManager	1	3	0	0	2	8