Incremental Testing of Object-Oriented Class Structures

Mary Jean Harrold and John D. McGregor
Clemson University

Abstract

Although there is much interest in creating libraries of well-designed, thoroughly-tested classes that can be confidently reused for many applications, few class testing techniques have been developed. In this paper, we present a class testing technique that exploits the hierarchical nature of the inheritance relation to test related groups of classes by reusing the testing information for a parent class to guide the testing of a subclass. We initially test base classes having no parents by designing a test suite that tests each member function individually and also tests the interactions among member functions. To design a test suite for a subclass, our algorithm incrementally updates the history of its parent to reflect both the modified, inherited attributes and the subclass's newly defined attributes. Only those new attributes or affected, inherited attributes are tested and the parent class's test suites are reused, if possible, for the testing. Inherited attributes are retested in their new context in a subclass by testing their interactions with the subclass's newly defined attributes. We have incorporated our class testing technique into Free Software Foundation, Inc's C++ compiler© and have used it in conjunction with a data flow tester for our experimentation.

Categories and Subject Descriptors: D.1.5[**Programming Techniques**]: Object-oriented Programming, D.2.5[**Software Engineering**]: Testing and Debugging

General Terms: Class, Incremental, Object-oriented, Testing

Additional Key Words and Phrases: Class Libraries, Class Testing, Object-oriented Testing

1. Introduction

One of the main benefits of object-oriented programming is that it facilitates reuse of instantiable, information-hiding modules, or *classes*. A class is a template that defines the *attributes* that an object of that class will possess. A class's attributes consist of (1) *data members* or *instance variables* that implement the object's state and (2) *member functions* or *methods* that implement the operations on the object's state. Classes are used to define new classes, or *subclasses*, through a relation known as *inheritance*. Inheritance imposes a hierarchical organization on the classes and permits a subclass to inherit attributes from its parent classes and extend, restrict, redefine or replace them in some way. A goal of object-oriented programming is to create libraries of well designed and thoroughly tested classes that can be confidently reused for many applications.

Authors' address: Department of Computer Science, Clemson University, Clemson, SC 29634-1906.
This work was partially supported by the National Science Foundation under Grant CCR-9109531 to Clemson University.

Although there is much interest in creating class libraries, few class testing techniques have been developed. One approach to testing class libraries is to validate each class in the library individually. This approach requires that each subclass be completely retested, although many of its attributes were previously tested since they are identical to those in a parent class. Additionally, completely retesting each class does not exploit opportunities to reuse and share design, construction and execution of test suites. Another approach to class testing is to utilize the hierarchical nature of classes related by inheritance to reduce the overhead of retesting each subclass. However, Perry and Kaiser[16] have shown that many inherited attributes in subclasses of well designed and thoroughly tested classes must be retested in the context of the subclasses. Thus, any subclass testing technique must ensure that this interaction of new attributes and inherited attributes is thoroughly tested. Fielder[4] presented a technique to test subclasses whose parent classes are thoroughly tested. Part of his test design phase is an analysis of the effects of inheritance on the subclass. He suggests that only minimal testing may be required for inherited member functions whose functionality has not changed. Cheatham and Mellinger[2] also discuss the problem of subclass testing and present a more extensive analysis of the retesting required for a subclass. However, both of these subclass testing techniques require that the analysis be performed by hand, which prohibits automating the design phase of testing. Additionally, neither technique attempts to reuse the parent class's test suite to test the subclass.

In this paper, we present an incremental class testing technique that exploits the hierarchical nature of the inheritance relation to test related groups of classes by reusing the testing information for a parent class† and incrementally updating it to guide testing of the subclass. We initially test *base classes* having no parents by designing a test suite that tests each member function individually and also tests the interactions among member functions. A *testing history* contains the test suites used for testing and associates each test case with the attributes that it tests. In addition to inheriting attributes from its parent, a newly defined subclass 'inherits' its parent's testing history. While a subclass is derived from its parent class, a subclass's testing history is derived from the testing history of its parent class. The inherited testing history is incrementally updated to reflect differences from the parent and the result is a testing history for the subclass. A subclass's testing history guides execution of the test cases since it indicates which test cases must be run to test the subclass. With this technique, we automatically identify new attributes in the

† Although a class may have several parents from which it can inherit attributes, for our discussion, we assume that each class has only one parent.

subclass that must be tested along with inherited attributes that must be retested. We retest inherited attributes in the context of the subclass by identifying and testing their interactions with newly defined attributes in the subclass. We also identify those test cases in the parent class's test suite that can be reused to validate the subclass and those attributes of the subclass that require new test cases.

The main benefit of this approach is that completely testing a subclass is avoided since the testing history of its parent class is reused to design a test suite for the subclass. Only new or replaced attributes in the subclass or those affected, inherited attributes are tested. Additionally, test cases from the test suite of the parent class are reused, if possible, to test the subclass. Thus, there is a savings in time to design new test cases, time to construct the test suite and actual time to execute the test cases since the entire subclass is not tested. Since our technique is automated, there is limited user intervention in the testing process. We have implemented our technique on a Sun-4 Workstation by incorporating it into Free Software Foundation, Inc's C++ compiler©, *g++*, where it is used in conjunction with a data flow tester. Our experiments on existing class hierarchies show the savings that can be realized using our technique.

The next section gives background information on procedural language testing since we apply similar testing techniques to class testing. Section 3 discusses inheritance in object-oriented programs as an incremental modification technique and defines the class attributes. Section 4 presents our incremental testing technique by first giving an overview, and then detailing, both base class testing and subclass testing. At the end of Section 4, we discuss our implementation. Section 5 discusses experimentation and concluding remarks are given in Section 6.

2. Testing

The overall goal of testing is to provide confidence in the correctness of a program. With testing, the only way to guarantee a program's correctness is to execute it on all possible inputs, which is usually impossible. Thus, systematic testing techniques generate a representative set of test cases to provide coverage of the program according to some selected criteria. There are two general forms of test case coverage: *specification-based* and *program-based*[9]. In specification-based or 'black-box' testing, test cases are generated to show that a program satisfies its

functional and performance specifications. Specification-based test cases are usually developed manually by considering a program's requirements. In program-based or 'white-box' testing, the program's implementation is used to select test cases to exercise certain aspects of the code such as all statements, branches, data dependences or paths. For program-based testing, analysis techniques are often automated. Since specification-based and program-based testing complement each other, both types are usually used to test a program.

While most systematic testing techniques are used to validate program *units*, such as procedures, additional testing is required when units are combined or integrated. For *integration* testing, the interface between units is the focus of the testing. Interface problems include errors in input/output format, incorrect sequencing of subroutine calls, and misunderstood entry or exit parameter values[1]. Although many integration testing techniques are specification-based, some interprocedural program-based testing techniques have recently been developed[7, 12].

A test set is *adequate* for a selected criterion if it covers the program according to that criterion [19] and a program is deemed to be *adequately* tested if it has been tested with an adequate test set. Weyuker[19] developed a set of axioms for test data adequacy that expose insufficiencies in program-based adequacy criteria. Several of these axioms are specifically related to unit and integration testing. The *antiextensionality* axiom reminds us that two programs that compute the same function may have entirely different implementations. While the same specification-based test cases may be used to test each of the programs, different program-based test cases may be required. Thus, changing a program's implementation may require additional test cases. The *antidecomposition* axiom tells us that adequately testing a program P does not imply that each component of P is adequately tested. Adequately testing each program component is especially important for those components that may be used in other environments where input values may differ. Thus, each unit that may be used in another environment must be individually tested. The *anticomposition* axiom tells us that adequately testing each component Q of a program does not imply that the program has been adequately tested. Thus, after each component is individually tested, the interactions among components must also be tested.

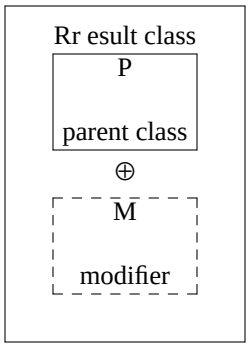

Figure 1. Inheritance as an incremental modification technique that uses the inheritance rules of the language to combine parent class P with modifier M to get subclass R. We use the operator ⊕ to symbolize this combination.

3. Inheritance in Object-Oriented Systems

Inheritance is a mechanism for both class specification and code sharing that supports developing new classes based on the implementation of existing classes. A subclass's definition is a *modifier* that defines attributes that alter the attributes in the parent class. The modifier and the parent class along with the inheritance rules for the language are used to define the subclass. The class designer controls the specification of the modifier while the inheritance relation controls the combination of the modifier and the parent class to get the subclass. Wegner and Zdonik[18] described inheritance as an incremental modification technique that combines a parent class P with modifier M to get a resulting class R. Figure 1 illustrates this incremental modification technique where we use the composition operator ⊕ to represent this uniting of M and P to get R, where R = P ⊕ M.

The subclass designer specifies the modifier, which may contain various types of attributes that alter the parent class to get the resulting subclass. We include the redefined, virtual and inherited† attributes presented by Wegner and Zdonik[18] and define an additional type of attribute, the *new* attribute. We further classify the virtual attribute as *virtual-new*, *virtual-inherited* and *virtual-redefined* to enable the identification of the required subclass (re)testing. An attribute is accessible *within* a class if the attribute is available to member functions in the class; an attribute is accessible *outside* a class if the attribute is available to users of the class. In the following list, we reference Figure 1, define the attributes and identify the scope to which they are bound.

† In their paper[18], Wegner and Zdonik use *recursive* instead of inherited to describe this type of attribute. For our purposes, we feel that ‘‘inherited’’ is more descriptive.

New attribute: (1) A is an attribute that is defined in M but not in P or (2) A is a member function attribute in M and P but A's signature† differs in M and P. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Inherited attribute: A is defined in P but not in M. In this case, A is bound to the locally defined attribute in P. A is accessible within R and accessible outside R if A is public; A is accessible both within and outside P.

Redefined attribute: A is defined in both P and M where A's signature is the same in M and P; we assume that the specification of A is the same in P and M. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Virtual-new attribute: (1) A is specified in M but its implementation may be incomplete in M to allow for later definitions or (2) A is specified in M and P and its implementation may be incomplete in P to allow for later definitions, but A's signature differs in M and P. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Virtual-inherited attribute: A is specified in P but its implementation may be incomplete in P to allow for later definitions, and A is not defined in M. In this case, A is bound to the locally defined attribute in P. A is accessible within R and accessible outside R if A is public; A is accessible both within and outside P.

Virtual-redefined attribute: A is specified in P but its implementation may be incomplete in P to allow for later definition, and A is defined in M with the same signature as A in P. In this case, A is bound to the locally defined attribute in M. A is accessible within R and outside R if A is public; A is not accessible in P.

Although modifier M transforms a parent class P into a resulting class R, M does not totally constrain R. We must also consider the inheritance relation since it determines the effects of composing the attributes of P and M and mapping them into R. The inheritance relation determines visibility, availability and format of P's attributes in R. A language may support more than one inheritance mapping by allowing specification of a parameter value to determine which mapping is used for a particular definition. For example, in C++, the *public*, *protected* and *private* keywords as part of the class specification determine visibility of attributes in the subclass. Since inheritance is deterministic, it permits the construction of rules to identify the availability and visibility of each attribute. This feature supports automating the process of analyzing a class definition and determining which attributes require testing.

To illustrate some of the different types of attributes, consider Figure 2, where class *P* is given on the left, the modifier that specifies *R*, a subclass of *P*, is given in the center, and the attributes for the resulting class *R* are given on the right. *P* has two data members, *i* and *j*, both integers, and four member functions, *A, B, C* and *D*. The modifier for class *R* contains one real data member, *i*, one constructor, *R*, and three member functions, *A, B* and *C*. The modifier is combined with *P* under the inheritance rules to get *R*. Data member float *i* is a new attribute in *R* since it

† The argument list is referred to as the *signature* of a function because the argument list is often used to distinguish one instance of a function from another[14].

```
class P {
  private:
     int i;
     int j;
  public:
     P( ){}
     void A(int a,int b)
          {i=a; j=a+2*b;}

     virtual int B( )
          {return i;}
     int C( )
          {return j;}
     int D( )
          {return B( );}
};
```

```
class R : public P {
  private:
     float i;

  public:
     R( ){}


     void A(float a)
          {i=a+4.5;}
     virtual int B( )
          {return 3*P::B( );}
     int C( )
          {return 2*P::C( );}


};
```

```
R's attributes after the mapping
  private:
     float i;                    //new

  public:
     R( ) {}                     //new
     void A(int a, int b)        //inherited
          {i=a; j=a+2*b;}
     void A(float a)             //new
          {i=a+4.5;}
     virtual int B( )            //virtual-redefined
          {return 3*P::B( );}
     int C( )                    //redefined
          {return 2*P::C( );}
     int D( )
          {return B( );}         //inherited
```

```
hidden
     int i;
     int j;
```

Figure 2. Class P on the left, subclass R's specification (modifier) in the center, and subclass R's attributes on the right.

does not appear in *P*. The constructor *R* is a new attribute in class *R*. Member function *A* that is defined in the modifier modifier, is a new attribute in *R* since its argument list does not agree with *A*'s argument list in *P*. Member function *A* in *P* is inherited in *R* since it is inherited unchanged from *P*. Thus, *R* contains two member functions named *A*. Member function *B* is virtual in *P* and since it is redefined in the modifier, it is virtual-redefined in *R*. Member function *C* is redefined in *R* since its implementation is changed by the modifier. Both member functions *B* and *C*, defined in *P*, are still accessible in *R* but only by member functions defined in *P*. Member function *D*, defined in *P*, is inherited in *R*. When *D* is called in *P*, it accesses *B*( ) in *P*; when *D* is called in *R*, it accesses *B*( ) in *R*. Finally, data members *i* and *j* in *P* are inherited but hidden† in *R*. Thus, they cannot be accessed by member functions defined in the modifier but are accessed by inherited member function *A*.

The modifier approach decomposes the inheritance structure into overlapping sets of class inheritance relations. The left side of Figure 3 shows a simple three-level chain of inheritance relations while the center illustrates an incremental view of the relationship among the classes. Class B can be replaced by A ⊕ M1 since A's attributes and M1's attributes are combined to form B. Once B is defined, there is no distinction in B between A's attributes

† An attribute belongs to the hidden area of a class if it is inherited from the private area of a parent class.

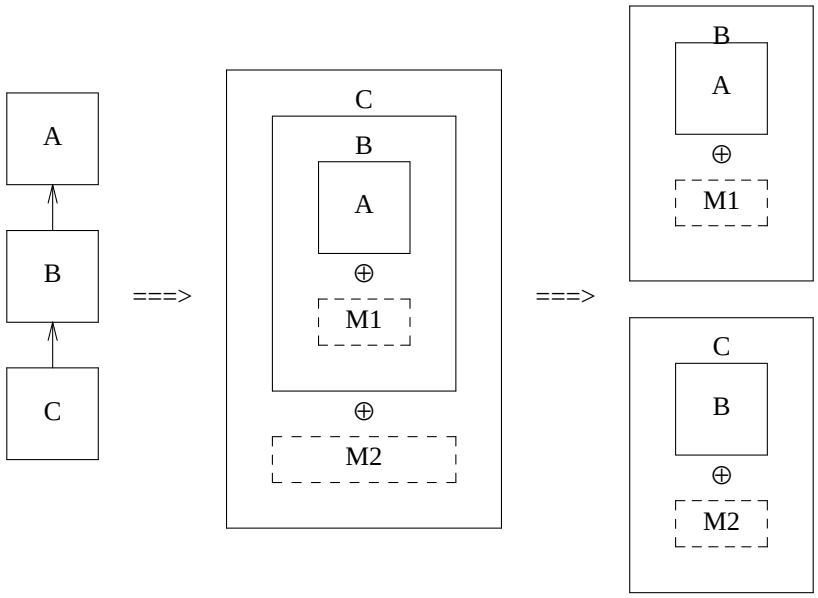

Figure 3. The inheritance hierarchy shown on the left indicates that A is a class with subclasses B and C, B is a class with subclass C. The figure in the center illustrates the incremental format for the class inheritance hierarchy where each new subclass is formed by combining the parent class with some modifier. The figure on the right shows how the class hierarchy can be decomposed into independent structures.

and M1's attributes. To define a subclass of B, the inheritance relation combines B and M2 in the same way. Thus, the three level inheritance relation can be decomposed into independent structures as illustrated on the right side of Figure 3. However, the inheritance relation imposes a partial order on the class resolutions in an inheritance structure. Class C can be determined without considering class A but the relation from A to B must be resolved prior to determining the relation from B to C. Thus, any inheritance structure can be decomposed into a set of partially ordered pairs of classes. This decomposition permits us to consider only a class definition and its immediate parents to fully constrain the definition of that class. This decomposition of class hierarchies also permits us to consider only the immediate parents and the modifier when testing a subclass.

4. Hierarchical Incremental Class Testing

Our class testing technique initially tests a base class by testing each member function individually and then testing the interactions among the member functions. The test suite, execution information and association between the member functions and test cases is saved in a testing history. Then, when a subclass is defined, the testing

history of its parent, the definition of the modifier and the inheritance mapping of the implementation language are used to derive the testing history for the subclass. The subclass's testing history indicates which attributes to (re)test in the subclass and which of the parent's test cases can be reused. Our technique is hierarchical because it is guided by the partial ordering of the inheritance relation; our technique is incremental because it uses the results from testing a class one level in the hierarchy to reduce the efforts needed by subsequent levels.

Our testing technique assumes a language model that is a generalization of the C++[17] model but is sufficiently flexible to support other languages with similar features such as Trellis[10]. Our language model is (1) strongly typed and permits polymorphic substitution to provide flexibility, (2) uses static binding whenever possible for efficiency, (3) supports three levels of attribute visibility with the same characteristics as C++'s *private*, *protected* and *public*, although the technique can handle any number of visibility levels, and (4) assumes a parameterized inheritance mapping with the two parametric values used in C++, *private* and *public*. The levels of visibility for attributes are ordered from most visible (*public*) to least visible (*private*) and the inheritance mapping maps an attribute to a level of visibility in the subclass that is at least as restrictive as its level in the parent class.

4.1. Base Class Testing

We first test base classes using traditional unit testing techniques to test individual member functions in the class. Our incremental testing technique addresses the test data adequacy concerns expressed by Perry and Kaiser[16]. The antidecomposition axiom tells us that adequate testing of the class does not guarantee adequate testing of each member function. Adequately testing each member function is particularly important since member functions may be inherited by the subclasses and expected to operate in a new context. Thus, we individually test each member function in a class using a test suite that contains both specification-based and program-based test cases. The specification-based test cases can be constructed using existing approaches such as the one proposed by Doong and Frankl[3]. The program-based test cases are constructed using existing techniques such as branch testing or data flow testing. The testing history for a class contains associations between each member function in the class and both a specification-based and a program-based test suite. Thus, the history contains triples, $\{m_i, (TS_i, \text{test?}), (TP_i, \text{test?})\}$ where m_i is the member function, TS_i is the specification-based test suite, TP_i is the program-based test suite and *test?* indicates whether the test suite is to be run to test the class.

The anticomposition axiom implies that testing each member function individually does not mean that the class has been adequately tested. Thus, in addition to testing each member function, we must test the interactions among member functions in the same class, *intra-class* testing; we must also test the interactions among member functions that access member functions in other classes, *inter-class* testing. Intra-class testing is guided by a *class graph* where each node represents either a member function in the class or a primitive data member, and each edge represents a message. For intra-class integration testing, we combine the attributes as indicated by the class graph and develop test cases that test their interfaces. For intra-class testing, we develop both specification-based and program-based test suites. The history for the class contains those member functions that call other member functions or access primitive data members. A member function that does not call other member functions has no integration test cases in the history but is integration tested with those member functions that call it. Thus, the second part of the history also consists of triples, $\{m_i,(TIS_i,test?),(TIP_i,test?)\}$ where m_i is a member function that calls other member functions in the class. TIS_i represents the specification-based integration test suite, TIP_i represents the program-based integration test suite and the *test?* field indicates if the test suite is to be totally (re)run (Y), partially (re)run (P), or not (re)run (N).

Inter-class testing is guided by interactions of the classes that result when member functions in one class interact with member functions in another class. Inter-class interactions occur when (1) a member function in one class is passed an instance of another class as a parameter and then sends that instance a message or (2) when an instance of one class is part of the representation of another class and then sends that instance as a message. The application's design provides a relationship among the class instances that is similar to the class graph produced for intra-class testing. The techniques for handling these interactions are like those described above for intra-class interactions except that interacting attributes are in different classes. In this paper, we focus on intra-class testing.

To illustrate our technique for testing base classes, consider the simplified example of a hierarchy of graphical shape classes implemented in C++[17]. Class *Shape*, given in Figure 4, is an abstract class that facilitates creation of classes of various shapes for graphics display. The class definition is abbreviated for purpose of illustration but the complete code is given in Appendices 1-3. Each 'shape' that can be drawn in the graphics system has a reference point that is used to locate the position where the shape is drawn in the program's coordinate system. The class graph for class *Shape* is also given in Figure 4. Rectangles represent member functions and ovals represent instances of classes. Solid lines indicate intra-class messages while dashed lines indicate inter-class messages.

```
class Shape {
private:
        Point reference_point;
public:
        void put_reference_point(Point new_point) {
           reference_point = newpoint(); }
        Point get_reference_point( ) {
           return reference_point; }
        void move_to(Point new_point) {
           erase( );
           put_reference_point(new_point);
           draw( ); }
        void erase( ) {
           draw( ); }
        virtual void draw( ) = 0;
        virtual float area( ) {
           cout << "Area: no reimplementation";
           return 0.0;}
        Shape(Point new_point) {
           put_reference_point(new_point); }
        Shape( ) { }
}
```

Figure 4. Definition for class *Shape* on the left and its class graph, showing the interacting member functions, on the right. The code for class *Shape* is given in Appendix 1.

Class *Shape* defines several member functions that describe the behavior of a shape. The only data member in *Shape* is *reference_point*, which is an instance of class *Point*. The *put_reference_point(Point)* and *get_reference_point( )* member functions provide controlled access to values of the data member, and thus, both interact with *reference_point*. The *move_to(Point)* member function is completely defined in terms of other member functions in class *Shape* even though some of these member functions are virtual and have no implementation. The *move_to(Point)* member function moves the shape to a new location. The *erase( )* member function is implemented with an ‘xor’ drawing mode so it only calls draw to overwrite, and thus erase, the existing figure. Class *Shape* has two virtual member functions, *area( )* and *draw( )*, that contribute to the common interface for all classes in the

Testing History for *Shape*				
attribute	specification-based test suite	program-based test suite	specification-based test suite	program-based test suite
	individual member functions		*interacting member functions*	
put_reference_point( )	(TS_1,Y)	(TP_1,Y)	(--)	(--)
get_reference_point( )	(TS_2,Y)	(TP_2,Y)	(--)	(--)
move_to( )	(TS_3,Y)	(TP_3,Y)	(TIS_3,Y)	(TIP_3,Y)
erase( )	(TS_4,Y)	(TP_4,Y)	(TIS_4,Y)	(TIP_4,Y)
draw( )	(TS_5,N)	(--)	(TIS_5,N)	(--)
area( )	(TS_6,Y)	(TP_6,Y)	(TIS_6,Y)	(--)
Shape( )	(TS_7,Y)	(TP_7,Y)	(TIS_7,Y)	(TIP_7,Y)
Shape( )	(TS_8,Y)	(TP_8,Y)	(--)	(--)

Table 1. Testing history for class *Shape* of Figure 4. 'Y' indicates that all test cases in test suite T_i are run while 'N' indicates that none of the test cases are run. The absence of a test suite is indicated by (--).

inheritance structure. Since *draw( )* is a pure virtual member function, it has an initial value of 0 and no implementation. Virtual member function *area( )* has an initial implementation that can be changed in subsequent subclass declarations. The *Shape(Point)* and *Shape( )* member functions are constructors of instances of the class.

Table 1 shows the testing history for class *Shape*. In Table 1, TS_i, TP_i, TIS_i and TIP_i represent the test suites used for the testing; a 'Y' or 'N' indicates whether the member function or interacting member functions are executed with the test case as input. The analysis of *Shape* is very straightforward. Since *Shape* is a base class, we test each of its available member functions. Thus, the specification-based and program-based test suites for *put_reference_point(Point)*, *get_reference_point( )*, *move_to(Point)*, *erase( )* and the two constructors, *Shape(Point)* and *Shape( )*, are generated. The specification-based test suite for *draw( )* can be generated but the test suite cannot be used as input since there is no implementation for *draw( )*. The program-based test suite for *draw( )* cannot be generated since no implementation exists. Since there is an initial implementation for *area( )*, both its specification-based and program-based test suites can be generated and run. The advantage of developing the specification-based test suites for *draw( )* and *area( )* in the base class is that these test suites can be 'inherited' and reused in the histories of subclasses.

Class *Shape*'s specification describes how individual member functions interact with each other. The input values that test these interactions belong to the specification-based integration test suites, TIS_i. According to the class specification, member functions *move_to(Point)*, *erase( )*, *draw( )*, *area( )*, and *Shape(Point)* call other class member functions. *move_to(Point)* calls *erase( )*, *draw( )* and *put_reference_point(Point)* while *erase( )* also calls *draw( )*. *draw( )* and *get_reference_point( )* both call *get_reference_point( )*, and *Shape(Point)* calls

put_reference_point(Point). The class graph for class *Shape*, shown in Figure 4, illustrates these interactions. Specification-based integration test cases are generated to test these interactions as shown in Table 1. For some of the interacting member functions, the program-based integration test cases can also be generated. For example, the interface between *move_to( )* and both *erase( )* and *draw( )* can be tested and TIP_4 contains these test cases. On the other hand, since *area( )* calls no other member functions in its implementation, no program-based integration test cases can be generated.

There are several inter-class messages such as the messages between *put_reference_point(Point)* and *reference_point*, which is an instance of class *Point*. Integration test cases are used to validate these messages but since our focus is intra-class testing, we omit them from this example.

4.2. Testing Subclasses

With each class in a hierarchy, we associate a testing history so that when a subclass is defined, our algorithm *TestSubclass* is used to derive the testing history for the subclass. *TestSubclass*, given in Figure 5, uses an incremental technique that transforms the testing history for the parent class P to the testing history for the subclass R. *TestSubclass* inputs P's history, HISTORY(P), P's class graph, G(P), and modifier M and outputs an updated HISTORY(R) for the subclass R. The actions taken by *TestSubclass* depend on the attribute type and the type of modification made to that attribute by the inheritance mapping. In Section 3, we discussed six types of attributes: new, inherited, redefined, virtual-new, virtual-inherited and virtual-redefined. For each of these types of attributes, different actions may occur. Algorithm *TestSubclass* begins by initializing R's testing history to that of its parent class P except that all *test?* fields in R's testing history are initially set to 'N'. Thus, we initially assume that there is no testing of the attributes in the subclass. Then, we update the history whenever we determine that testing is required. The algorithm then inspects each attribute *A* in the modifier M, and takes appropriate action to update R's testing history and determine the required testing.

If *A* is a NEW or a VIRTUAL-NEW member function, it must be tested completely tested since it was not defined in P. We thoroughly test *A* individually so that when *A* is inherited by some subclass, only integration testing will need to be repeated. Since the anticomposition axiom tells us that it is necessary to retest each new member function in its new context, *A* must be integration tested with other member functions in R with which it interacts. If *A* is NEW, it has both a specification and an implementation, so we generate new specification-based

```
algorithm TestSubclass(HISTORY(P),G(P),M);

input:   HISTORY(P):P's testing history; G(P):P's class graph;
         M:modifier that specifies subclass R;
output:  HISTORY(R):testing history for R indicating what to rerun;
         G(R):class graph for subclass R;

begin
  HISTORY(R) := HISTORY(P);                                  /* initialize R's history to that of P */
  foreach attribute in HISTORY(R) do test? := 'N'            /* reset all test? fields */
  G(R) := G(P);                                              /* initialize R's class graph to that of P */
  foreach attribute A ∈ M do
    case A is NEW:                                           /* A is a new attribute */
        Generate TS , TP for A;                              /* unit test A */
        Add {A, (TS, 'Y'), (TP, 'Y')} to HISTORY(R);         /* add TS, TP to testing history */
        Integrate A into G(R);                               /* identify attributes with which A interacts */
        Generate any new TIS and TIP;                        /* new test cases to test interaction */
        Add {A, (TIS, 'Y'), (TIP, 'Y')} to HISTORY(R);       /* update history to reflect changes */
    case A is VIRTUAL-NEW:                                   /* A is a virtual-new attribute */
        Generate TS for A;                                   /* since A has an implementation */
        if A is defined then
           Generate TP for A;                                /* since A has an implementation */
           Add {A,(TS, 'Y'),(TP, 'Y')} to HISTORY(R);        /* update history to reflect changes */
           Integrate A into G(R);                            /* identify attributes with which A interacts */
           Add {A,(TIS ,'Y'), (TIP, 'Y')} to HISTORY(R);     /* update history to reflect changes */
        else                                                 /* A has no implementation */
           Add {A,(TS,'N'), ( -- )} to HISTORY(R);           /* update history to reflect changes */
           Integrate A into G(R);                            /* identify attributes with which A interacts */
           Add {A,( -- ), ( -- )} to HISTORY(R);             /* update history to reflect changes */
        endif
    case A is REDEFINED or VIRTUAL-REDEFINED:                /* A is redefined /redefined-virtual */
        Generate TP for A;                                   /* since A has a new implementation */
        Add {A, (TP, 'Y')} to HISTORY(R);                    /* update history to reflect changes */
        Reuse TS from P if it exists or Generate TS for A;   /* try to reuse test cases from parent */
        Update {A, (TS, 'Y')} in HISTORY(R);                 /* update history to reflect changes */
        Integrate A into G(R);                               /* identify attributes with which A interacts */
        Generate TIP for G(R)  with respect to A;            /* new test cases to test interaction */
        Reuse TIS from P;                                    /* no new spce-based required */
        Update {A, (TIS, 'Y/P'), (TIP, 'Y/P')} in HISTORY(R); /* update history to reflect changes */
end TestSubclass.
```

Figure 5. Algorithm *TestSubclass* that determines a testing HISTORY(R) by incrementally updating testing HISTORY(P). HISTORY(R) is used to test the subclass.

and program-based test suites to unit test *A*. Both of these new test suites, TS and TP, are marked for execution by setting *test?* to 'Y' and HISTORY(R) is updated to reflect the change. *A* is integrated into the class graph G(R) and any new integration specification-based and program-based test suites are generated and HISTORY(R) is updated. If *A* is VIRTUAL-NEW, we generate specification-based test cases to unit test *A*. If *A* is defined, we also generate new program-based test cases for unit testing. In either case, HISTORY(R) is updated to reflect the new test suites.

A is then integrated into G(R). If *A* is defined, integration test cases are generated and HISTORY(R) is updated; otherwise, no integration test cases are generated.

A REDEFINED or VIRTUAL-REDEFINED attribute *A* in M also requires extensive retesting but many existing specification-based test cases may be reused since only the implementation has changed. The antiextensionality axiom tells us that since the implementation has changed, new program-based test cases may be required. If *A* is a data member (i.e., an instance of a class) we assume that the class to which the instance belongs has been tested. No other individual testing is performed on *A* although it may be involved in the integration testing of the member functions defined in M. If *A* is a member function, the specification of *A* remains unchanged but the implementation of *A* will have changed. Thus, new program-based test cases, TP, are generated for *A* and added to HISTORY(R). The specification-based test cases stored in HISTORY(R) for the previous definition of *A* are still valid and are reused; HISTORY(R) is marked to rerun these test cases. If *A* is redefined, it is individually retested by generating new program-based test suites to test the implementation of *A*. HISTORY(R) is updated to reflect the new test suites and reused existing test cases, and these test cases are marked for reusing by setting *test?* to 'Y'. Then, *A* is integrated into G(R). New program-based interface test cases, TIP, are generated and marked for testing by setting *test?* to 'Y' or 'P'. 'Y' indicates that the entire test suite is reused while 'P' indicates that only the test cases identified as testing affected parts of the subclass are reused.

A INHERITED or VIRTUAL-INHERITED member function attribute *A* requires very limited retesting since it was previously tested individually in parent class P and its specification and implementation remain unchanged. Thus, the specification-based and program-based test suites for *A* are not rerun. Integration test cases are not reused if they only test the interaction of *A* with other inherited attributes since this interaction has also been previously tested. However, *A* may interact with new or redefined attributes or access the same instances in the class's representation as other member functions. Although *TestSubclass* does not take any action when a INHERITED or VIRTUAL-INHERITED attribute *A* is encountered, the antidecomposition axiom reminds us that it is necessary to test *A* in its new context in the subclass. *A*'s interaction with new or redefined attributes is tested when those member functions are integrated into the subclass.

This limited testing adequately tests the attributes in the subclass because of the extensive testing that occurred when the attribute was defined. To see this, consider the case in which a inherited member function accesses the same data as a new attribute. The inherited member function was tested in P and the specification of the inherited member function remains unchanged. Thus, specification-based and program-based test cases need not be rerun. The only test cases that are rerun are those that test the interactions between *A* and any new or redefined member function(s). These test cases are identified by *TestSubclass* when it handles the NEW or REDEFINED member functions.

```
class Triangle: public Shape {
        private:
                Point vertex2;
                Point vertex3;
        public:
                Point get_vertex1( );           //new
                Point get_vertex2( );           //new
                Point get_vertex3( );           //new
                void put_vertex1(Point);        //new
                void put_vertex2(Point);        //new
                void put_vertex3(Point);        //new
                void draw( );                   //virtual-redefined
                float area( );                  //virtual-redefined
                Triangle( );                    //new
                Triangle(Point,Point,Point);    //new
}
```

Figure 6. Definition of class *Triangle*, a subclass of class *Shape*. A comment with each of the public attributes indicates its type; all other inherited attributes are inherited. The code for class *Triangle* is given in Appendix 2.

Both data member and member function attributes are defined in the parent class and inherited by the subclass. The inheritance mapping from P into R may change the visibility of a data member attribute. For example, if the attribute has moved from a visible level to one that is not visible then it cannot interact with any new or redefined attribute. The data attributes that are hidden and the member functions on that data form a tested unit that need not be retested. If the data attribute is visible to any new member functions that are defined in R, then the interfaces between the new member functions and the existing member functions that access the data attributes must be tested. This testing is performed when a new member function that interacts with data attributes accessed by existing member functions is integrated into G(R).

To illustrate the way in which algorithm *TestSubclass* works, we consider subclasses of class *Shape*, whose code is given in Figure 4. First, consider the definition of class *Triangle*, given in Figure 6, that is a subclass of class *Shape*. Class *Triangle* contains data and member functions to provide more specific information about the 'shape' of the object such as data for the additional points needed for the vertices of the triangle and a new implementation of member function *area( )* to find the area of a triangle. The benefits of hierarchical incremental testing can be seen in the testing history for class *Triangle*, given in Table 2. Since *put_reference_point (Point)*, *get_reference_point( )*, *move_to(Point)* and *erase( )* are inherited attributes, none of the test suites for the individual member functions are rerun. Virtual-redefined member functions *draw( )* and *area( )* are retested since they have new implementations. However, only new program-based test cases are developed since existing specification-based test cases

Testing History for *Triangle*				
attribute	specification-based test suite	program-based test suite	specification-based test suite	program-based test suite
	individual member functions		*interacting member functions*	
put_reference_point( )	(TS_1,N)	(TP_1,N)	(--)	(--)
get_reference_point	(TS_2,N)	(TP_2,N)	(--)	(--)
move_to( )	(TS_3,N)	(TP_3,N)	(TIS_3,P)	(TIP''_3,P)
erase( )	(TS_4,N)	(TP_4,N)	(TIS_4,P)	(TIP''_4,P)
draw( )	(TS_5,Y)	(TP'_5,Y)	(TIS_5,P)	(TIP'_5,Y)
area( )	(TS_6,Y)	(TP'_6,Y)	(TIS_6,Y)	(TIP''_6,P)
Shape( )	(TS_7,N)	(TP_7,N)	(TIS_7,N)	(TIP_7,N)
Shape( )	(TS_8,N)	(TP_8,N)	(--)	(--)
get_vertex1( )	(TS'_9,Y)	(TP'_9,Y)	(TIS'_9,Y)	(TIP'_9,Y)
get_vertex2( )	(TS'_{10},Y)	(TP'_{10},Y)	(TIS_{10},Y)	(TIP'_{10},Y)
get_vertex3( )	(TS'_{11},Y)	(TP'_{11},Y)	(TIS'_{11},Y)	(TIP'_{11},Y
put_vertex1( )	(TS'_{12},Y)	(TP'_{12},Y)	TIS'_{12},Y)	(TIP'_{12},Y)
put_vertex2( )	(TS'_{13},Y)	(TP'_{13},Y)	(TIS'_{13},Y)	(TIP'_{13},Y)
put_vertex3( )	(TS'_{14},Y)	(TP'_{14},Y)	(TIS'_{14},Y)	(TIP'_{14},Y)
Triangle( )	(TS'_{15},Y)	(TP'_{15},Y)	(TIS'_{15},Y)	(TIP'_{15},Y)
Triangle( )	(TS'_{16},Y)	(TP'_{16},Y)	(--)	(--)

Table 2. Testing history for Class *Triangle* of Figure 6. 'Y' indicates that all test cases in the test suite are run, 'P' indicates that some of the test cases are run and 'N' indicates that none of the test cases are run. The absence of a test suite is indicated by (--). New test suites are marked with ´ and those test suites that are changed are marked with ´´.

can be reused. Test suites are developed and run for all new member functions defined in class *Triangle*. There are several interactions that must be tested. When virtual-redefined member function *draw( )* is encountered, it is integrated into G(R) and interacts with *move_to( ), erase( )* and *get_reference_point( )*. Although neither *move_to( )* nor *erase( )* has changed, both of them are tested with the new implementation of *draw( )*. Some of the existing specification-based test cases are reused in the testing. New program-based test cases may be required to test the interface with the new implementation of *draw( )*. The actual test cases in the test suite that are required depend on the type of testing being used; we indicate that some test cases must be rerun with 'P' in the *test?* field. The specification-based test suite for the interaction of *draw( )* and *get_reference_point( )* must be rerun and new program-based integration test cases must be generated and run. A similar situation exists for the retesting of virtual-redefined member function *area( )* except that some existing program-based test cases may be reused. No new or redefined attributes interact with either of the *Shape* constructors, so no additional interface testing is required for *Shape( )* or *Shape(Point)*. All new attributes in class *Triangle* except constructor *Triangle( )* interact with other attributes and thus, new test cases are generated and run to interface test them. In Table 2, test suites marked with ´ are newly developed, test suites marked with ´´ may have newly developed test cases, while all others are reused

from the parent.

```
class EquiTriangle: public Triangle{
    public:
        float area( );                          //redefined
        Equi_triangle(Point,Point,Point);       //new
        Equi_triangle( );                       //new
}
```

Figure 7. Definition of class *EquiTriangle*, a subclass of both *Shape* and *Triangle*. A comment with each of the public attributes indicates its type; all other inherited attributes are inherited. The code for class *EquiTriangle* is given in Appendix 3.

The last class in the hierarchy is class *EquiTriangle*, a subclass of both class *Shape* and class *Triangle*, that adds no new member functions other than the constructors for the class. However, *EquiTriangle* redefines the implementation of *area( )* to provide more efficiency. Only the program-based test cases for *area( )* are regenerated, although all test cases for *area( )* are rerun. Integration test cases to test the interactions of the new and redefined member functions with the inherited attributes are also run. The definition of class *EquiTriangle* is given in Figure 7 and its testing history is given in Table 3. Since we use an incremental technique to generate the history for a subclass, *TestSubclass* only requires the immediate parent of *EquiTriangle*, or *Triangle* to determine what must be tested.

Testing History for *EquiTriangle*				
attribute	specification-based test suite	program-based test suite	specification-based test suite	program-based test suite
	individual member functions		*interacting member functions*	
put_reference_point( )	(TS_1,N)	(TP_1,N)	(--)	(--)
get_reference_point( )	(TS_2,N)	(TP_2,N)	(--)	(--)
move_to( )	(TS_3,N)	(TP_3,N)	(TIS_3,N)	(TIP_3,N)
erase( )	(TS_4,N)	(TP_4,N)	(TIS_4,N)	(TIP_4,N)
draw( )	(TS_5,N)	(TP_5,N)	(TIS_5,N)	(TIP_5,N)
area( )	(TS_6,Y)	(TP'_6,Y)	(TIS_6,P)	(TIP''_6,Y)
Shape( )	(TS_7,N)	(TP_7,N)	(TIS_7,N)	(TIP_7,N)
Shape( )	(TS_8,N)	(TP_8,N)	(--)	(--)
get_vertex1( )	(TS_9,N)	(TP_9,N)	(TIS_9,N)	(TIP_9,N)
get_vertex2( )	(TS_{10},N)	(TP_{10},N)	(TIS_{10},N)	(TIP_{10},N)
get_vertex3( )	(TS_{11},N)	(TP_{11},N)	(TIS_{11},N))	(TIP_{11},N)
put_vertex1( )	(TS_{12},N)	(TP_{12},N)	((TIS_{12},N)	((TIP_{12},N)
put_vertex2( )	(TS_{13},N)	(TP_{13},N)	(TIS_{13},N)	(TIP_{13},N)
put_vertex3( )	(TS_{14},N)	(TP_{14},N)	(TIS_{14},N)	(TIP_{14},N)
Triangle( )	(TS_{15},N)	(TP_{15},N)	(TIS_{15},N)	(TIP_{15},N)
Triangle( )	(TS_{16},N)	(TP_{16},N)	(--)	(--)
Equi_triangle( )	(TS_{17},Y)	(TP_{17},Y)	(TIS_{17},Y)	(TIP_{17},Y)
Equi_triangle( )	(TS_{28},Y)	(TP_{28},Y)	(--)	(--)

Table 3. Definition and History for Class *EquiTriangle*. Test suites marked with ‘Y’ or ‘P’ are reused to test the subclass; those marked with ‘N’ are not rerun. ‘Y’ indicates that all test cases in the test suite are reused; ‘P’ means that only part of that test suite is reused. New test suites are marked with ´ and those test suites that are changed are marked with ´´.

4.3. Implementation

The implementation of our testing system consists of two main parts. The first part of our testing system is the *History Generator* that uses our algorithm *TestSubclass* to automatically identify the required retesting in a subclass. The second part of our testing system uses a data flow tester to assist in performing the testing. Both are incorporated into the GNU *g++* compiler.

We altered the parsing phase of *g++* to get our *History Generator* that transforms the history of the parent class to that of the subclass. The *History Generator* inputs the parent class, the history for the parent class and the definition of the subclass and outputs the history for the subclass.

Although, our hierarchical incremental algorithm is independent of the testing methodology, we used a type of program-based testing known as data flow testing[5, 11, 15] to demonstrate the feasibility of our technique. We modified the intraprocedural data flow analysis performed by the g++ compiler[8] to gather the definition-use pairs for the testing. Another technique [6, 7] uses this intraprocedural data flow information to compute the interprocedural definition-use pairs. Thus, we perform intraprocedural data flow testing on individual member functions and interprocedural data flow testing on interacting member functions.

5. Experimentation

We are using a variety of existing C++ class hierarchies for our experimentation to determine the savings in testing gained using our technique. We have tested the class hierarchies in InterViews 3.0.1 [13], which is a library of graphics interface classes. A representative class hierarchy in the InterViews class library, shown in Figure 8, has class *Resource* as its base class. Here, we report the results for the hierarchy with base class *Resource* and subclasses, *Glyph, Interactor, Scene, MonoScene* and *Dialog*. Table 4 shows statistics about the classes in this hierarchy including the number of lines of code in each class, and the numbers and types of each member function attribute.

We used our *History Generator* to determine which of the methods in each class required retesting. The results of that analysis are also shown in Tables 5, 6 and 7. We compared our technique with a technique that retests all methods since there are no other existing incremental methods. The results show that for this particular path, in one hierarchy, a significant amount of effort is saved with our technique. Table 5 shows the retesting required for

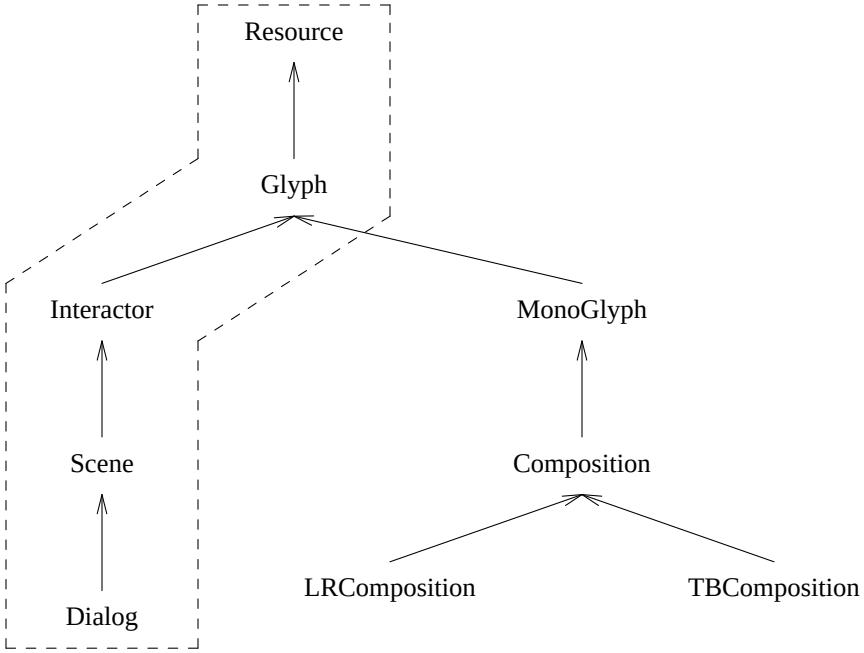

Figure 8. Class hierarchy of *InterViews* 3.0.1 rooted at class *Resource*. The results reported in Tables 5-7 are for part of the class hierarchy enclosed in the dashed box.

the specification-based test suites for the classes in the hierarchy. For the individual member in Table 6, the retesting for the program-based test suites is given. For the individual member functions, the number of member functions that are tested generally decreases as we descend in the hierarchy. This decrease is expected for a well-designed hierarchy with a large amount of functionality defined at the top level, and with modifications and additions made at the lower levels.

Member functions to be tested (specification-based)			
class	*retest all*	*our method*	*ours/retest all*
Resource	9	9	100%
Glyph	25	16	64%
Interactor	68	43	63%
Scene	84	16	19%
MonoScene	85	1	1%
Dialog	90	5	6%

Table 5. Specification-based testing results using *History Generator* for hierarchy rooted at base class *Resource*.

Member functions to be tested (program-based)			
class	*retest all*	*our method*	*ours/retest all*
Resource	9	9	100%
Glyph	25	17	68%
Interactor	68	44	65%
Scene	84	19	23%
MonoScene	85	9	11%
Dialog	90	7	8%

Table 6. Program-based testing results using *History Generator* for hierarchy rooted at base class *Resource*.

We combined the results for both the specification-based and program-based integration test suites in Table 7. For the integration test suites, the testing depends on the interactions of the member functions and while there is a savings in the required testing, the number of functions to be tested does not generally decrease as we descend the hierarchy.

Member functions to be tested (interaction)			
class	*retest all*	*our method*	*ours/retest all*
Resource	9	9	100%
Glyph	25	1	4%
Interactor	68	15	22%
Scene	84	15	18%
MonoScene	85	3	4%
Dialog	90	10	11%

Table 7. Interaction testing results using *History Generator* for hierarchy rooted at base class *Resource*.

This analysis did not consider another potential benefit of the technique; the benefits derived from reuse of the parent tests suites. Many of the methods that must be retested will reuse the specification-based test cases that were developed for their parent class. Reusing test suites from the parent class results in substantial additional savings of time for the testing process.

6. Conclusion

We have presented an incremental technique to assist in testing classes that exploits the hierarchical structure of groups of classes related by inheritance. Our language model is a generalization of the C++ [17] language. Base classes are initially tested using both specification-based and program-based test cases, and a history of the testing information is saved. A subclass is then tested by incrementally updating the history of the parent class to reflect the

differences from the parent. Only new attributes or those inherited, affected attributes and their interactions are tested. The benefit of this technique is that it provides a savings both in the time to analyze the class to determine what must be tested and in the time to execute test cases. We initially incorporated data flow testing into our hierarchical testing system for both individual member functions and interacting member functions to provide base class testing and subclass testing. Later, we will include a specification-based testing technique in the testing system.

References

1. B. Beizer, in *Software Testing Techniques*, Van Nostrand Reinhold Company, Inc., New York, 1990.
2. T. J. Cheatham and L. Mellinger, ‘‘Testing object-oriented software systems,’’ *Proceedings of the 1990 Computer Science Conference*, pp. 161-165, 1990.
3. R-K. Doong and P. G. Frankl, ‘‘Case studies on testing object-oriented programs,’’ *Proceedings of the Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pp. 165-177, October 1991.
4. S. P. Fielder, ‘‘Object-oriented unit testing,’’ *Hewlett-Packard Journal*, pp. 69-74, April 1989.
5. P. G. Frankl and E. J. Weyuker, ‘‘An applicable family of data flow testing criteria,’’ *IEEE Transactions on Software Engineering*, vol. SE-14, no. 10, pp. 1483-1498, October 1988.
6. M. J. Harrold and M. L. Soffa, ‘‘Interprocedural data flow testing,’’ *Proceedings of the Third Testing, Analysis, and Verification Symposium (TAV3 - SIGSOFT89)*, pp. 158-167 , Key West, FL , December 1989.
7. M. J. Harrold and M. L. Soffa, ‘‘Selecting Data for Integration Testing,’’ *IEEE Software, special issue on testing and debugging*, pp. 58-65, March 1991.
8. M. J. Harrold and P. Kolte, ‘‘Combat: A compiler based data flow testing system,’’ *Proceedings of Pacific Northwest Quality Assurance*, pp. 311-323, October 1992.
9. W. E. Howden, in *Software Engineering and Technology: Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.
10. M. Killian, ‘‘Trellis: Turning designs into programs,’’ *CACM*, vol. 33, no. 9, pp. 65-67, September 1990.
11. B. Korel and J. Laski, ‘‘A tool for data flow oriented program testing,’’ *ACM Softfair Proceedings*, pp. 35-37, December 1985.
12. U. Linnenkugel and M. Mullerburg, ‘‘Test data selection criteria for integration testing,’’ *Proceedings of the 1990 Conference on Systems Integration*, pp. 45-58, April 1990.
13. M. A. Linton and P. R. Calder, ‘‘The design and implementation of InterViews,’’ *Proceedings of USNIX C++ Workship*, pp. 256-267, 1987.
14. S. B. Lippman, in *C++ Primer, Second Edition*, p. 121, Addison-Wesley Publishing Company , New York, 1991.
15. S. C. Ntafos, ‘‘An evaluation of required element testing strategies,’’ *Proceedings of 7th International Conference on Software Engineering*, pp. 250-256, March 1984.
16. D. E. Perry and G. E. Kaiser, ‘‘Adequate testing and object-oriented programming,’’ *Journal of Object-Oriented Programming*, vol. 2, pp. 13-19, January/February 1990.
17. B. Stroustrup, in *The C++ Programming Language*, Addison-Wesley Publishing Company, Massachusetts, 1986.
18. P. Wegner and S. B. Zdonik, ‘‘Inheritance as an incremental modification mechanism or what like is and isn’t like,’’ *Proceedings of ECOOP’88*, pp. 55-77, Springer-Verlag, 1988.
19. E. J. Weyuker, ‘‘Axiomatizing software test data adequacy,’’ *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, pp. 1128-1138, December 1986.

Appendix 1: Source Code for Class *Shape*

```
#ifndef DISPLAYH
#define DISPLAYH

#include "point.h"

class display{
  public:
        display();
        clear();
        line(Point, Point);
        line_to(Point);

};
#endif
=======================================================
#ifndef POINTH
#define POINTH

class Point{
  private:
        int x;
        int y;
  public:
        void put_x(int);
        void put_y(int);
        int get_x();
        int get_y();
        Point();
        Point(int,int);
};
#endif
=======================================================
#ifndef SHAPEH
#define SHAPEH

#include "point.h"

class Shape{
  private:
        Point reference_point;
  public:
        void put_reference_point(Point);
        Point get_reference_point();
        void move_to(Point);
        void erase();
        virtual void draw()=0;
        virtual float area();
        Shape(Point);
        Shape();
};
#endif
```

```
#include "iostream.h"
#include "shape.h"
#include "gsystem.h"


void Shape::put_reference_point(Point new_point){
  reference_point = new_point;
}

Point Shape::get_reference_point(){
  return reference_point;
}

void Shape::move_to(Point new_point){
  erase();
  put_reference_point(new_point);
  draw();
}

void Shape::erase(){
//Assumes XOR drawing mode
  draw();
}

float Shape::area(){
  cout << "Area:  no reimplementation";
  return 0.0;
}

Shape::Shape(Point new_point){
  put_reference_point(new_point);
}

Shape::Shape(){

}
```

Appendix 2: Source Code for Class *Triangle*

```
#ifndef TRIANGLEH
#define TRIANGLEH

#include "shape.h"

class Triangle:public Shape{
  private:
        Point vertex2;
        Point vertex3;

  public:
        Point get_vertex1();
        Point get_vertex2();
        Point get_vertex3();
        void put_vertex1(Point);
        void put_vertex2(Point);
        void put_vertex3(Point);
        void draw();
        float area();

        Triangle(Point,Point,Point);
        Triangle();
};

#endif
```

```
#include "shape.h"
#include "triangle.h"
#include "display.h"
#include "gsystem.h"


Point Triangle::get_vertex1(){
  return get_reference_point();
};

Point Triangle::get_vertex2(){
  return vertex2;
};

Point Triangle::get_vertex3(){
  return vertex3;
};

void Triangle::put_vertex1(Point new_point){
  put_reference_point(new_point);
};

void Triangle::put_vertex2(Point new_point){
  vertex2 = new_point;
};

void Triangle::put_vertex3(Point new_point){
  vertex3 = new_point;
};

void Triangle::draw(){
  //commands to a global terminal to draw lines
  terminal.line(get_vertex1(),get_vertex2());
  terminal.line(get_vertex2(),get_vertex3());
  terminal.line(get_vertex3(),get_vertex1());
};

float Triangle::area(){
   Point p1,p2,p3;
   p1 = get_vertex1();
   p2 = get_vertex2();
   p3 = get_vertex3();
   return abs(0.5*(p1.get_x()*p2.get_y()+p1.get_y()*p3.get_x()+p3.get_y()*
   p2.get_x()- p2.get_y()*p3.get_x()-p1.get_y()*p2.get_x()-p1.get_x()*
   p3.get_y()));
};

Triangle::Triangle(Point new_point1,Point new_point2,Point new_point3):
(new_point1){
  put_vertex2(new_point2);
  put_vertex3(new_point3);
};

Triangle::Triangle():(){};
```

Appendix 3: Source Code for Class *EquiTriangle*

```
#ifndef EQUIH
#define EQUIH

#include "triangle.h"

class EquiTriangle:public Triangle{
  public:
        float area();
        EquiTriangle(Point,Point,Point);
        EquiTriangle();
};

#endif

=======================================================

#include "equitriangle.h"


float EquiTriangle::area(){
   Point p1, p2, p3;
        p1 = get_vertex1();
        p2 = get_vertex2();
        p3 = get_vertex3();
        float s1 = sqrt(sqr(p2.get_x() - p3.get_x()) - sqr(p2.get_y() - p3.get_y()));
        float s2 = sqrt(sqr(p1.get_x() - p2.get_x()) - sqr(p1.get_y() - p2.get_y()));
        return .433*s1*s2;
};

EquiTriangle::EquiTriangle(Point new_point1,Point new_point2,Point new_point3)
:(new_point1,new_point2,new_point3){};

EquiTriangle::EquiTriangle():(){};
```