The Effectiveness of Category-Partition Testing of Object-Oriented Software

Alisa Irvine and A. Jefferson Offutt*
ISSE Department
George Mason University
Fairfax, VA 22030
phone: 703-993-1654
fax: 703-993-1638
email: ofut@isse.gmu.edu

March 3, 1995

Abstract

When migrating from conventional to object-oriented programming, developers face difficult decisions in modifying their development process to best use the new technology. In particular, ensuring that the software is highly reliable in this new environment poses different challenges. Developers need understanding of effective ways to test the software. This paper presents empirical data that show that the existing technique of category-partition testing can effectively find faults in object-oriented software, and new techniques are not necessarily needed. For this study, we identified types of faults that are common to C++ software and inserted faults of these types into two C++ programs. Test cases generated using the category-partition method were used to test the programs. A fault was considered detected if it caused the program to terminate abnormally or if the output was different from the output of the original program. The results show that the combination of the category-partition method and a tool for detecting memory management faults may be effective for testing C++ programs in general. Since there is no evidence that traditional techniques are not effective, software developers may not need new testing methods when migrating to object-oriented development.

1 Introduction

Although a significant amount of research has been accomplished in object-oriented analysis, design and programming, very little has been done in object-oriented software testing. We are interested in whether traditional software testing techniques are effective for testing object-oriented software.

*Partially supported by the National Science Foundation under grant CCR-93-11967.

We chose to focus on specification-based testing of object-oriented software. We used the category-partition method because we felt this technique was appropriate for object-oriented software. To measure the effectiveness of the category-partition method for testing object-oriented software, faults unique to object-oriented software were inserted into C++ programs. For our purposes, *effectiveness* is a measure of the fault detection ability of a testing technique [FW91].

The rest of Section 1 provides a brief introduction to the basic concepts of object-oriented programming and software testing. Section 2 surveys seven papers that have been published on this subject and relates their positions on two questions: *How can we use the properties of object-oriented software to reduce the effort required to test object-oriented programs?* and *How can we effectively test object-oriented programs?* The latter question involves two issues: whether traditional techniques are effective for object-oriented software and whether new techniques need to be developed. We focused on the first of these issues. Section 3 describes our approach to measuring the effectiveness of the category-partition method for testing C++ programs and our results. Section 4 discusses conclusions and Section 5 considers future work.

1.1 Introduction to Object-Oriented Concepts

Object-oriented programs model objects in the real world to solve problems. An object-oriented program consists of a number of objects, each exhibiting behavior, having a state and an identity [Boo91]. These objects work together to perform the actions required of the program.

A *class* packages data with functions that may operate on that data. An *object* is a specific instance of a class. For example, Cat is a class; Fluffy is an instance of the class, or, in other words, Fluffy is an object that is a member of the Cat class. A *method* or *member function* of a class defines how the objects belonging to that class will *behave*: Fluffy, Mittens and Snowball all eat, purr, sleep and pounce in the same way. A *client* is an object that uses the resources of another by calling its member functions. An object has *state*, which encompasses all of the properties of an object and all of the values of those properties [Boo91]. For example, Fluffy weighs 1.2 pounds and has black fur; Mittens weighs 2.5 pounds and is brown with white paws. Each object also has *identity*, meaning that we can distinguish among the objects of a class. In our feline example, the names that we have given our objects indicate their identity: Mittens is distinguishable from Snowball. The terms *function, member function* (from C++) and *method* (from Smalltalk) are used interchangeably to refer to "operations that a client may perform upon an object" [Boo91].

Two key concepts of object-oriented programming are encapsulation and inheritance. *Encapsulation*, also known as *data hiding*, prevents clients from knowing about or depending on the implementation of a class. For example, a Stack class is defined to have the methods `push(item)`, `pop()` and `numElements()`. This interface is the only way that the elements stored in the stack may be accessed by a user of this class. An implication of this is that the Stack may be originally implemented using an array and later changed to a linked list. Any code that uses the Stack cannot depend on the implementation, and therefore, will not have to be modified.

Encapsulation allows classes to be defined outside of the context of a particular program. The Stack class described above provides the ability to store objects and retrieve them in a Last-In-First-Out ordering. Any program requiring that capability is able to use the Stack class. In this way, encapsulation promotes reuse of classes.

Inheritance allows common features of many classes to be defined in one class. Then other classes (derived classes) may be defined by taking the features they need from the existing class (base class). These features may be enhanced or restricted in the derived class. For example, an AirlineTicket has certain properties: it may be purchased and refunded; it keeps track of the purchaser, the flight number, the seat number, the date of the flight and the cost. A SuperSaverTicket has these methods and the same data, but modifies them slightly. For example, the ticket must be purchased more than 21 days before the flight and cannot be refunded fewer than 14 days from the flight. The cost of the ticket is also significantly lower. In this example, the AirlineTicket is the base class and the SuperSaverTicket is a derived class. By allowing classes to share features, inheritance also promotes software reuse.

The terms *parent class* and *superclass* are equivalent and refer to "the class from which another inherits" [Boo91]. *Base class* is similar, referring to "the most generalized class in a class structure" [Boo91]. The terms *child class, derived class* and *subclass* are also equivalent and refer to "a class that inherits from one or more classes" [Boo91].

Parameterized classes, called *generics* in Ada and *templates* in C++, are another, somewhat less common, feature of object-oriented programs. A parameterized class provides a capability that is not dependent on a specific type. For example, a LinkedList could be written as a parameterized class. At the time of instantiation of a LinkedList, inventory_list, the type of objects that it stores, InventoryItem, would be specified:

```
LinkedList   inventory_list<InventoryItem>;
```

Since the capability need only be written once but may be used with many different types, parameterization is a third way that object-oriented programming promotes software reuse.

Different languages may support different features, but in order to be called object-oriented, a language must meet minimum requirements. An *object-oriented programming language* must [CW85]:

1. support programming with abstract data types and information hiding
2. associate a type (class) with an object, and
3. support inheritance.

According to these requirements, Smalltalk, C++, and Ada9X are object-oriented languages. All three support single inheritance; C++ and Ada9X also support multiple inheritance. Ada83 is referred to as an *object-based language* since it does not support inheritance [CW85], but it does support encapsulation and parameterization.

Object-oriented programs can be used to model real world objects, and thus are considered to be easier for a programmer to understand and maintain [CM90, SR90]. In addition, encapsulation should localize modifications, making maintenance easier [CM90].

Consider a common example, a graphics program. This simplified system draws, moves and rotates circles and squares. An object-oriented design for this system might define an abstract base class, `Shape`, that will define the feature that are common to circles and squares (and any geometric shapes to be added at a later date). It might also derive two classes, `Circle` and `Square`, from `Shape` to define those features they do not have in common. For example, they all have a reference point; they all may be told to draw themselves, move themselves or rotate themselves. In some cases, these functions may be done the same way, so these are defined in `Shape`. For example, all shapes move by adding the given offsets to their reference point and re-drawing themselves. In other cases, these functions may be done differently, depending on the specific shape. These would be defined in the derived class. For example, a `Circle` draws itself differently than a `Square` does. Figure 1 shows an object-oriented design for this hierarchy, using Coad's notation [CE91]. In C++, these classes might be implemented as shown in Figure 2. (*draw_square()* and *draw_circle()* are functions provided by a separate graphics package.)

C++ has some specialized kinds of functions. *Constructors* are responsible for allocating any memory needed and initializing all data members. The object should be in a valid state when the

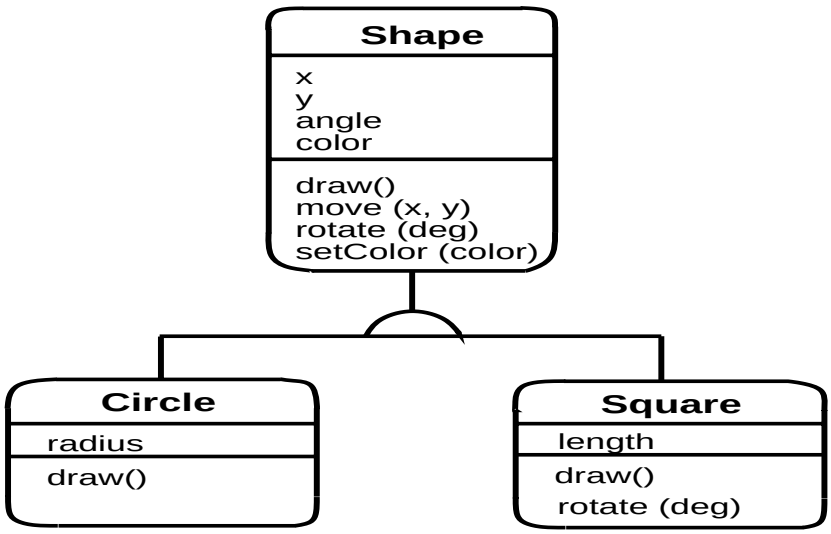

Figure 1: **Object-oriented design of Shape hierarchy**

constructor exits. A constructor is automatically called when an object is defined, when an object is passed by value into a function or when a function returns an object. Also, a constructor is called explicitly by the `new` operator.

Destructors are responsible for cleaning up (including de-allocating memory) when an object is destroyed. An object is destroyed either implicitly or explicitly. If the object is a data member of another class, it is automatically destroyed when the destructor of the object it is a part of is executed. If the object is local to a function, it is automatically destroyed when control of the program leaves the function. If the object was created dynamically, it is destroyed when the `delete` operator is explicitly called with it as the parameter. A virtual destructor will automatically call the destructors of all parent classes.

In C++, different functions may be defined with the same name if they have different functional prototypes. This capability, called *function overloading*, allows functions that perform the same conceptual task within a class to be defined with the same name. For example, a String class could have two insert functions, one that takes an integer i and a character and the other that takes an integer i and a String object. The first function inserts the character at location i and the second inserts the string at location i. Operators such as +, =, ==, etc. may also be overloaded. This capability should not be confused with *polymorphism*, which allows a subclass to redefine a function inherited from its parent. From the graphics example above, the `rotate` function was defined in a generic way in the base class, `Shape`, then redefined as necessary in derived classes, such as `Square`.

```
class Shape {
     public;
      Shape (int new_x, int new_y, Color clr)
        : x(new_x), y(new_y), color(clr) { }
      ~ Shape  ();
      virtual void draw ( )      { }
      void move {int x_offset, int y_offset) { x += x_offset;
                                               y += y_offset;
                                               draw (); }
      virtual void rotate (float degrees)    { }
      void setColor (Color new_color)      { color = new_color; }
     protected:
      int x;
      int y;
      float angle;
      Color color;
};


class Circle :: public Shape {
    public:
      Circle(int new_x, int new_y, float rad, Color clr)
        : x(new_x), y(new_y), radius(rad), color(clr)   { }
      ~ Circle ();
      virtual void draw ( )      { draw_circle(x, y, radius, color); }
   protected:
     float radius;
};


class Square :: public Shape {
   public:
     Square(int new_x, int new_y, float len, Color clr)
       : x(new_x), y(new_y), length(len), color(clr)    { }
     ~Square ( );
     virtual void draw ( )      { draw_square(x, y, length, angle, color); }
     virtual void rotate (float degrees)    { angle += degrees;
                                              draw( ); }
   protected:
     float length;
};
```

Figure 2: **C++ code for graphic example**

Polymorphism may be thought of as *replacing* a function, where overloading *adds* a function. Also, polymorphism involves a parent and a child class, where overloading only involves one class.

The C++ compiler will generate certain functions if they are not defined explicitly. If no constructor is defined, one will be defined that takes no parameters. A non-virtual destructor will be defined if the class' parent defines a destructor. A copy constructor, an assignment operator and two address-of operators will also be defined implicitly if they are not defined explicitly. If the class is at all complex (for instance, if it dynamically allocates memory), these implicit functions will probably not perform as desired.

1.2 Introduction to Testing Concepts

Software testing techniques are roughly divided into two categories: white box and black box [Whi87]. White box techniques are also known as structural techniques, because they explicitly use the structure of the program to generate test data. Black box techniques, which are also known as functional testing techniques, generate test data with no knowledge of the code or the structure of the software.

Examples of white box testing methods are *path analysis, data flow testing, domain testing*, and *mutation testing*. The path analysis strategies [How76] select a set of paths to execute during testing, and test cases are derived to cause these paths to be executed. A set of coverage measures have been defined to indicate the extent of coverage. For example, *statement coverage* indicates that every statement has been executed at least once; *branch coverage* indicates that every conditional has evaluated to true and false at least once. Data flow testing [RW82, FW88] defines a new set of criteria based on data flow analysis [AC76], which determines the definition-use relationships of variables in a program. Data flow testing then uses this information to determine if a program has satisfied a certain testing criterion. For example, the *all-definitions* criterion requires that every global definition is used. The *all-uses* criterion requires that a path from every global definition to each of its uses is executed by the test set. The *all-du-paths* criterion is satisfied if a path from every definition of every variable to every use of the variable is executed. If there are multiple paths from a definition to a use, they must all be executed, up to but not including loops. Domain testing [WC80] involves partitioning the input domain into segments determined by the predicates in the path condition. Then the values on and around the border are tested, since it has been shown that these points are most sensitive to domain errors. Mutation testing [DLS78] produces a large number of "mutant" programs, each of which has a small modification from the original

program, as defined by one of a set of mutation operators. The test data is then run against these mutant programs to see how well the test data performs at detecting the modifications. The test set is sufficient when all mutant programs have been detected by the test set or determined to be functionally equivalent to the original program.

Examples of black box testing methods are *functional testing, specification-based testing*, and *category-partition testing*. The goal of functional testing [How85] is to identify the requirements or specifications of the functions to be tested and to derive the test data from these. Test cases focus on boundary conditions, special cases, error handlers and cases that are potentially dangerous [OB88]. Specification-based tests [Whi87] are a specialized type of functional test, where test cases are generated from formal or informal function specifications. The input domain of the function is partitioned into equivalence classes, where one value in a class is essentially the same as any other value in that class for testing purposes. Then at least one value is selected from each equivalence class to generate the test data. A weakness of functional testing, and specifically, specification-based testing, is that there is no well-defined method of partitioning the input domain [OB88]. The category-partition method was developed in response to this weakness. It defines a systematic way to develop test specifications and test cases. For this project, the category-partition method was used to generate the test cases, so a more detailed description of this method follows.

1.3 The Category-Partition Method

In general terms, the category-partition method [OB88] identifies those elements that influence a function and generates test cases by methodically varying these elements over all values of interest. The steps of this method are summarized below.

1. Determine the functional units to be tested and identify the parameters and environmental conditions that affect each unit. Then determine the *categories*, which are the major characteristics of the input domain of the function under test. For example, a *copy_file* function may take a file name and a directory name as its parameters and the environment could represent the current state of the file system. Categories of this function could include the validity of the file name and the validity of the directory name. Categories may include preconditions if formal methods were used to write the specifications. In the *copy_file* example, one category might reflect a precondition regarding the existence of the given file in the specified directory.

2. Partition the categories into *choices*, where each choice represents an equivalence class - a set of values that are considered to be the same for testing purposes. By definition, choices in each category must be disjoint, and together, the choices in each category must cover the input domain [AO94]. The choices for the File Existence Precondition might be:

   (a) No file or directory with the given file name exists in the specified directory.

   (b) A directory with the given file name exists in the specified directory.

   (c) A file with the given file name already exists in the specified directory.

   The structure of the code, if known, may influence the selection of categories.

3. Determine constraints among the choices, including restrictions on particular choices or in cases where a combination of choices might be impossible or undesirable.

4. Write the formal *test specification* for this function, indicating the parameters, environmental conditions, categories, the choices for each category, and the constraints on the choices. From this test specification, an automated tool may be used to generate *test frames*, which consist of a set of choices from the specification.

5. Evaluate test frames and determine if any changes are required. For example, look for test situations that need to be added or constraints that should be placed on a choice.

6. Convert test frames to *test cases* by selecting values to satisfy all the choices. Finally, write *test scripts* for the tet cases. Test scripts set up appropriate environmental conditions and parameters, run the test case, verify the result and clean up as needed.

1.4 A Refinement of the Category-Partition Method

The category-partition work has left a considerable amount of detail in steps 4-6, generating test scripts from test specifications, to the discretion of the tester. In particular, which combinations of categories to use is an important problem whose solution affects the strength and efficiency of testing. One problem solved by this refinement is that some of the combinations of categories are impossible, because they have conflicting requirements. Thus these combinations must be recognized and avoided. Another problem resolved is that the number of combinations of categories can be quite large and repetitious.

A refinement of the category-partition method [AO94] isolates those tasks of producing a test specification that are mechanical. The following seven steps may be performed to generate test scripts from test specifications.

1. Create an N-dimensional matrix, where N is the number of categories, that will represent all possible combinations of choices of categories. Entries in the matrix will specify a corresponding test frame.
2. Identify a *base test frame* by designating one default (normal) choice for each category.
3. Choose other combinations as test frames by combining each choice in a category with the base choice for all other relevant categories. This causes each non-base choice to be used at least once, and the base choices to be used several times. More combinations may be chosen by the test engineer as desired.
4. Identify infeasible combinations and determine a feasible combination by varying other choices until a possible combination is found.
5. Refine test frames into test cases by selecting a value for each choice.
6. Write operation commands, setup commands, verify commands and cleanup commands.
7. Create test scripts by combining the appropriate setup script, the operation command, verify command and cleanup command.

As an example, suppose the following was the test specification for a function `pattern_match` that takes two parameters: a pattern of up to 5 characters to search for and an expression of unlimited length in which to search for it.

Functional Unit: `pattern_match`
Inputs: `p? : String`
`e? : String`
Env. Variables: `None`
Categories: `Type of p?:`
`Choice 1: len(p?) >= 1 ∧ len(p?) < 5`
`Choice 2: len(p?) = 0`
`Choice 3: len(p?) = 5`

```
   Choice 4:    len(p?)  > 5
Type of e?:
   Choice 1:    e?  contains p?
   Choice 2:    e?  does not contain p?  ∧ len(e?)  = 0
   Choice 3:    e?  does not contain p?  ∧ len(e?)  > 0
```

Step 1: The combination matrix for this example will be a 2-dimensional matrix (because there are 2 categories) with dimensions 4x3 (because there are 4 choices for one category and 3 for the other). This is shown in Figure 3.

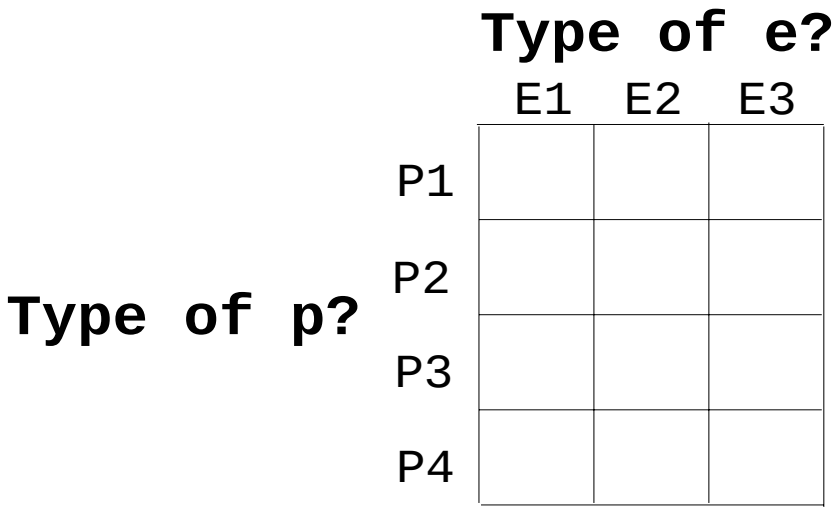

Figure 3: **Example of a combination matrix**

Step 2: The normal choice for each category will be the first choice. The base test frame will be the combination of (P1, E1).

Step 3: The other combinations chosen are shown in Figure 4. The choice of (P2, E2) was added at the tester's discretion.

Step 4: The combination (P2, E1) is not a feasible combination, since an expression cannot contain an empty pattern. E1 is the normal choice, so this is the choice varied. (P2,E3) is not already selected and is feasible, so this combination will be substituted.

Step 5: Test cases are chosen for the test frames:

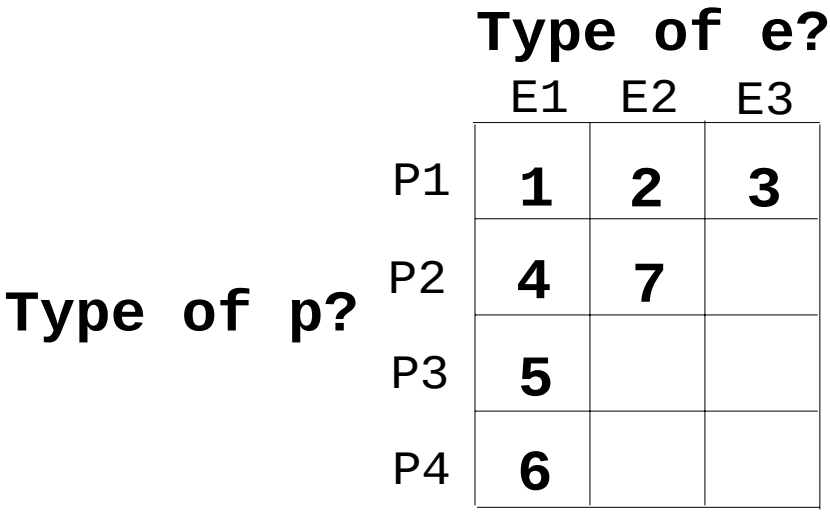

Figure 4: **Example of choosing combinations**

Test case 1: p? = "pat"; e? = "pattern"
Test case 2: p? = "pat"; e? = ""
Test case 3: p? = "pat"; e? = "category"
Test case 4: p? = ""; e? = "pattern"
Test case 5: p? = "other"; e? = "another test"
Test case 6: p? = "testing"; e? = "testing is fun"
Test case 7: p? = ""; e? = ""

Steps 6 and 7, writing commands and setting up test scripts, are not specific to category-partition, so they will not be shown here.

There are several benefits of this refinement. It frees the test engineer for more intellectually demanding tasks. It also provides a method of resolving infeasible tests caused by conflicting choices. Using formal specifications reduces the amount of effort required, since less effort is required to produce test specifications from formal specifications. A favorable side effect is that sometimes anomalies may be uncovered in the functional specifications.

2 Previous Work in Object-Oriented Software Testing

There are two general questions that are the focus of current object-oriented software testing research: *How can we use the properties of object-oriented software to reduce the effort required to test object-oriented programs?* and *How can we effectively test object-oriented programs?* The first question follows the intuitive notion that inheritance, encapsulation and parameterized classes should reduce the effort involved in testing software. The second question involves two issues: whether traditional techniques are effective for object-oriented software and whether new techniques need to be developed. In this section, seven papers are briefly presented and the positions expressed in each on the questions above are described.

2.1 Fiedler

Fiedler [Fie89] describes a testing methodology that was applied to a small set of generic classes. First, specification-based tests were performed by the development team. Then path analysis testing was done by an independent team to satisfy the all-branches criterion, and extra test cases were added based on testers' experience. These extra tests exercised object-oriented features: constructors, destructors, initialization of data members, casting and combinations of member functions. The methodologies of equivalence partitioning and boundary value analysis were also used in the generation of test cases. Defects were found by the white box technique that had not been uncovered by the black box technique.

Fiedler states that effort can be saved by using the inheritance property: "Provided that the functionality of the base class has been proven, any member function of the target class that leverages directly from a base class member function will require minimal testing," which is later described as "a basic functionality test" [Fie89]. There is no evidence given in the paper to support this claim.

Although the results from Fiedler's study indicate that specification-based testing is not effective for testing object-oriented software, it should be noted that the development team itself performed the specification-based tests. Software developers are notoriously bad at finding defects in software they have developed [Bei90].

2.2 Perry and Kaiser

Perry and Kaiser [PK90] disagree with the intuitive notion that classes may be reused without re-testing in the new, derived context. They feel that most inherited code must be re-tested. This theory is proven by applying Weyuker's test adequacy axioms [Wey88] to object-oriented features. Perry and Kaiser believe that object-oriented techniques promote reuse, but do not always aid the testing phase. They do not discuss the effectiveness of traditional testing techniques on object-oriented programs.

2.3 Cheatham and Mellinger

Cheatham and Mellinger [CM90] address the impact of object-oriented properties on unit- and system-level testing of object-oriented software. They claim that inheritance reduces the amount of testing required during unit-testing. If the member function being tested is derived unaltered, little additional testing is needed - only the interface needs to be re-tested. If the member function modifies the function from which it is derived, then the parent's version may be used for black box testing. If the member function completely replaces the function from which it is derived or is not related to existing functions, then it must be re-tested as a new member. No evidence is given to support this claim.

While they do not address specific testing methodologies, Cheatham and Mellinger state that white box techniques are appropriate for member functions, and black box testing may be done against the requirements and interface descriptions for each class. It is not clear from these descriptions to what extent testing is necessary nor which testing methods are effective.

2.4 Smith and Robson

Smith and Robson [SR90] discuss the problems of using current testing techniques for object-oriented programming systems. They do not attempt to define solutions, but emphasize that new techniques must be developed and suggest areas where further research is needed.

One reason that testing object-oriented software is more difficult is that the elements cannot be dynamically tested directly. A class does not actually exist - an instance of the class, an object, exists and is testable. Parameterized classes also cannot be tested directly because a parameter

must be replaced by an actual type. A third example is abstract base classes, which are designed to allow subclasses to inherit the same features. Because these features are left to the subclasses to define, they are not implemented in the abstract base class. These types of classes cannot be instantiated (because they are not fully defined), and may only be tested indirectly.

The manner in which classes evolve also causes testing problems. A change to a parent class can potentially affect all descendants. These changes may require significant revisions to tests and standard (traditional) regression testing techniques may not apply.

Traditional methods may not be effective with object-oriented programs. For example, the data flow analysis approach is questionable for object-oriented software since the flow through object-oriented code is much more complex than traditional programs.

Smith and Robson divide inheritance into four categories: simple, simple non-strict, multiple and repeated. The issue of minimizing effort in testing is not discussed directly, but they do indicate that when testing classes of the first category, the test cases of the parent class may be reused.

2.5 Doong and Frankl

Doong and Frankl [DF91] describe a new approach to testing object-oriented programs. This new method, which is based on the theory of algebraic specifications, focuses on the state into which a message or sequence of messages puts an object. Correctness is tested by determining if two sequences put an object into the same state, i.e., if it is impossible to distinguish between the two results by applying methods of the class (or related classes). Their research also focuses on automation of the testing process: test and test driver generation, test execution and test results hecking.

Case studies done using this technique show that the property of inheritance can reduce effort in testing. In most cases, the derived class should be tested against its own specification and the specification of the parent class for methods that have been modified by the derived class. There are exceptions to this, when it is not necessary for a subclass to conform to its parent's specification. When testing a subclass against its own specification, it is only necessary to test methods that are new or that interact with methods from the parent class.

Since the goal of their new technique is to reduce the effort in testing, specific testing methods and their effectiveness with object-oriented software are not discussed in their paper. Specification-

based tests were used and it was noted that program-based testing may be useful.

2.6 Harrold and McGregor

Harrold and McGregor [HM92] have developed an incremental technique for testing that reduces that amount of testing needed by exploiting inheritance relationships among classes. The axioms from Perry and Kaiser [PK90] are used to determine which functions need to be re-tested in the context of the subclass and which may inherit the test results from the parent class. Multiple inheritance is not discussed in the method.

Each class has a testing history associated with it. Base classes have a test suite designed that tests (using both specification-based and program-based tests) each member function and the interactions among the member functions. Derived classes inherit the base class' test history, then the test history is updated to reflect differences between the base and derived classes. In this step, the derived class' member functions are identified as fitting into one of six categories: *new, inherited, redefined, virtual-new, virtual-inherited* and *virtual-redefined*. Based upon the category, different levels of re-testing must be done. Experimental results show that sometimes significant savings in testing effort may be achieved.

The effectiveness of traditional testing techniques was not discussed. Although the incremental technique is independent of the testing methodology used, it was noted that data flow testing was used in their experiments.

2.7 Turner and Robson

Turner and Robson [TR93] introduce state-based testing, outline a process for testing object-oriented programs and describe a set of tools developed for assisting with state-based testing. The goal of state-based testing is to detect when a function changes the state of the object to an undefined or inappropriate state, which includes not changing the state when it is expected. The state of an object is the combination of the values of all data members at any given time.

Turner and Robson claim that specification-based tests and structural tests, while necessary, are not sufficient for testing object-oriented programs. Unfortunately, there is no evidence given to support this. Specification-based tests validate the external view of the clas. Structural tests are required to ensure effective coverage. State-based tests emphasize that the methods correctly modify the

class' data members. State-based tests are most effective with classes that have a high degree of interaction with their data members. It was noted that classes developed for the set of tools were undergoing testing using these three techniques, but the results were not available in their paper.

2.8 Summary

For the first question that we posed in Section 2, *How can we use the properties of object-oriented software to reduce the effort required to test object-oriented programs?*, there is agreement that effort in testing may be reduced by taking advantage of inheritance relationships where member functions are inherited unchanged from the base class. The extent of effort that may be saved and the best technique to use are not yet clear.

The second question we posed in Section 2 was *How can we effectively test object-oriented programs?* We focused on the first issue of this question: whether traditional techniques are effective for object-oriented software. None of these seven papers directly addresses this issue. Three papers [PK90, DF91, HM92] did not approach the issue. Cheatham and Mellinger [CM90] claim that traditional white box and black box techniques are effective, although they have no evidence to support this. Of the other three [Fie89, SR90, TR93] who do not believe traditional techniques (alone) to be effective, only Fiedler has any evidence, and that evidence is inconclusive. It should be noted that Turner and Robson do not reject traditional techniques for object-oriented software; they just do not believe that they are thorough enough. The technique they are developing is for use in addition to traditional techniques.

The second issue was whether new techniques need to be developed to effectively test object-oriented software. Although 3 papers describe new techniques being developed, 2 of these are aimed at reducing the effort involved in testing [DF91, HM92]. Only Turner and Robson [TR93] are developing a technique to improve the effectiveness of testing object-oriented software. Again, they showed no evidence that adding their technique improves testing results.

3 Do Traditional Testing Techniques Effectively Test Object-Oriented Software?

White box and black box techniques are quite different. White box techniques use the source code for generating test cases. Black box techniques generate test cases based on information external to

the source code, for example, requirements and function specifiations. Because of these differences, the effectiveness of white box and black box techniques need to be considered separately. We chose to focus on black box techniques, specifically the category-partition method, because we felt this technique was appropriate for object-oriented software. C++ was chosen as the programming language because it is becoming one of the most widely used object-oriented programming languages. Since category-partition is a specification-based technique, the programming language used should not be significant.

3.1 The Approach

To measure the effectiveness of the category-partition method in detecting faults inherent to object-oriented C++ programs, twenty-three types of faults were identified and two object-oriented programs were chosen to insert faults of these types into. Test cases were generated using the category-partition method with the refinements described in Section 1.4. The faults were inserted into the two programs, and they were then run with the test cases. If the program crashed or if the output was different from the output of the original program, the fault was considered to have been detected.

3.1.1 Types of Faults

Two sources were used to identify types of faults that are unique to object-oriented C++ programs: a book that describes common mistakes that are made with C++ programs [Mey92] and practical experience from 6 years of developing software using C++. Twenty-three types of faults were identified: 20 from the book of mistakes, 3 from experience. The fault types considered were divided into five categories: memory management, implicit functions, initialization, inheritance and encapsulation

A program exhibits memory management faults when it improperly allocates or releases memory when creating or destroying objects. There are two typical problems that occur: dangling references and memory leaks. A *dangling reference* is created when memory storage is freed while there are still active pointers or references to it [WSHF81]. A *memory leak* occurs when there exists memory storage that is allocated but inaccessible. In some programming languages, such as PASCAL, this is referred to as *garbage*, and *garbage collection* may be performed by the programming system to reclaim this memory space [WSHF81]. The detection of garbage requires significant execution

time, however, and is not done in C++.

Implicit functions are those functions that C++ compilers will generate automatically if they are not written explicitly: a copy constructor, an assignment operator, two address-of operators, a default constructor and a destructor. Initialization faults occur after an object has been created but before the constructor runs. The initialization list follows the single colon between the constructor's function prototype and the code (see the Shape constructor in Figure 2). Inheritance faults are related to improper design or implementation of an inheritance hierarchy. Encapsulation faults violate the principle of encapsulation, or information hiding.

For our purposes, a *fault* is defined to be a mistake that will result in a failure on certain inputs. A fault may consist of multiple pieces, or potential faults. A *potential fault* is defined to be a characteristic of the program that will result in a fault only if certain other characteristics also appear. For example, a pointer being assigned a value of NULL is a potential fault; it is part of a fault only if the pointer may be dereferenced later during execution. For example, a class String contains a potential fault: one function returns a pointer to internal data. A program using this class only exhibits the fault if the other two pieces of the fault also exist: the object's internal data is changed by using the pointer and the String object is used again. We were careful as we were inserting faults into the programs to include all the pieces of the faults.

The potential fault types are listed below. First, the potential fault type is briefly explained, then any other characteristics required to create a complete fault are described.

Memory management:

1. **Use new and free with a built-in type. (mm-nf-builtin)**
2. **Use malloc and delete with a built-in type. (mm-md-builtin)**
3. **Use new and free with an object. (mm-nf-object)**
   Note: The combination of `malloc` and `delete` with an object is not feasible, since there is no way to initialize the objects after the `malloc`.

   These three types of potential faults mix the use of `malloc` and `free` with `new` and `delete`. The Annotated Reference Manual [ES90], which defines the C++ language, states that these functions must never be mixed, even for built-in types. The results of doing so are undefined.

   `Malloc` and `free` have a different purpose from `new` and `delete`, and they are not interchangeable. `Malloc` allocates a specified amount of space, but does not run constructors as

`new` does. `Free` deallocates the specified space, but does not run destructors as `delete` does.

Characteristics:

Fault Type 1:

(a) A variable of a built-in type is created using `new`.

(b) The variable is destroyed using `free`.

Fault Type 2:

(a) A variable of a built-in type is created using `malloc`.

(b) The variable is destroyed using `delete`.

Fault Type 3:

(a) An object is created using `new`.

(b) The object has dynamically allocated memory (that is deleted in the destructor). This will cause a memory leak.

(c) The object is destroyed using `free`.

4. **Allocate a single object using new, destroy it using delete[]. (mm-del-arr)**
This type of potential fault uses inconsistent forms of the `new` and `delete` functions. `New` allocates space for the specified number of objects, a single object or an array of objects, and runs constructors for each. `Delete` will run destructors and deallocate the space, but it must be told whether the pointer it is given refers to a single object or an array. Adding the notation "`[]`" tells the compiler that the pointer refers to an array of objects. If the pointer refers to a single object, `delete` should be called; if it refers to an array of objects, `delete[]` should be called.

   This type of potential fault allocates a single object, but then attempts to delete an array of objects. The opposite, creating an array of objects and using `delete`, is also a potential fault, since deleting a single object when an array was allocated creates a memory leak. This second type of potential fault was not used in this project because an array of objects could not be created naturally.

   Characteristics:

   (a) A single object is allocated using `new`.

   (b) The object is deleted using `delete []`.

5. **Neglect to delete a pointer data member in a destructor. (mm-no-del)**
   If a data member of an object points to memory it allocated dynamically, not deleting that memory in the destructor will cause a memory leak.

   Characteristics:

   (a) The object dynamically allocates memory.

   (b) A pointer to this memory is not deleted in the destructor.

6. **Return a reference to a local variable. (mm-ret-local-ref)**
   The gnu compiler gives a warning about this potential fault type. The warning is quite specific, but a new user might not understand its significance. When the function returns, the local variable is destroyed, and the reference that is returned points to a non-existent object. A failure may result if the reference is used.

   Characteristics:

   (a) A function that returns a reference is defined.

   (b) Within that function, a local object is defined.

   (c) From that function, a reference to this local object is returned.

   (d) The reference that was returned is used.

7. **Return a reference to an object created by new in that function. (mm-ret-new-ref)**
   The object created by new, as any dynamically allocated memory, must be deleted somewhere in the program or a memory leak will result. Clearly, this object is not intended to be deleted inside the same function it was created in, since a reference to the object is returned. This means that the caller of the function is expected to delete the object. The caller must take the address of the result of the function in order to delete the object. This is not likely to happen in every case, and the result is a memory leak.

   A second scenario is even worse: this function call, which may be an operator, might be embedded in other function calls. This means that no reference to the object is saved, so it *cannot* be deleted. This will always be a memory leak.

   Characteristics:

   (a) A function that returns a reference is defined.

   (b) Within that function, an object is created using `new`.

(c) From that function, a reference to the new object is returned.

(d) The address of the reference returned is not deleted.

Or,

(a) This function or operator call is embedded in another function call.

Implicit functions:

8. **For a class that dynamically allocates memory, neglect to create a copy constructor. (impl-no-cc)**

9. **For a class that dynamically allocates memory, neglect to create an assignment operator. (impl-no-op=)**

   Neglecting to define a copy constructor or an assignment operator is probably incorrect if the class dynamically allocates memory [Mey92]. If a copy constructor (or assignment operator) is not defined, the compiler will generate one that will copy (or assign) each data member using that member's copy constructor (or assignment operator). Built-in types are copied (or assigned) bit-wise. This means that if an object with no copy constructor (or assignment operator) has a pointer to dynamically allocated memory, and this object is copied (or assigned) to another object, these two objects will refer to the same memory space. Since the destructor for the class should delete the memory allocated by the object (see #5), when either of these objects is destroyed, this memory should be deleted. The remaining object is then left with a dangling pointer, which references freed memory. A failure may result if this pointer is used.

   Characteristics:

   (a) The class allocates memory dynamically.

   (b) A copy constructor/assignment operator is not defined.

   (c) The object copied/assigned from deletes or modifies its dynamically allocated memory.

   (d) The object copied/assigned to uses that memory again.

   Or,

   (e) The object copied/assigned to deletes or modifies its dynamically allocated memory.

   (f) The object copied/assigned from uses that memory again.

10. **Make a base class destructor non-virtual. (impl-nonvirt-destr)**

If the base class destructor is non-virtual, then only the base class' destructor will be invoked when an object of the derived class is destroyed. Any cleaning up performed by the derived class' destructor will not be done because the destructor will never be called.

Characteristics:

(a) The base class destructor is defined to be non-virtual.

(b) An object that is a member of the derived class is created and dynamically allocates memory.

11. **Neglect an assignment to a data member within an assignment operator. (impl-msng=-op=)**

This type of potential fault may result in a failure if the assignment operator is used and the data member that did not receive a value is also used.

Characteristics:

(a) An assignment to a data member is neglected.

(b) The assignment operator must be called.

(c) The neglected data member must be used after the assignment.

12. **Duplicate the name of a data member. (impl-dup-name)**

If the name of a data member of an object is duplicated in a function, the data member will be hidden and cannot be referenced in that function. Thus, when the object by that name is given a value, the local object will be modified, rather than the object's data member. This may result in a failure if the object is used later.

Characteristics:

(a) A local object is defined with the same name and class as a data member.

(b) An operation is performed on the local object (that is intended for data member).

(c) The data member is used later.

Initialization:

13. **If the initial value of a data member depends on the value of another data member, declare the dependent data member first. (init-dep-member)**

Data members are created and initialized in the order of their declaration in the class so that the destructor can be assured to delete the data members in the opposite order from which

they were created. If a data member is dependent on the value of another data member, but the dependent data member is declared first in the header file, then it will not be initialized with the proper value [Mey92]. A failure may result if the dependent data member is used later, before its value is set.

Characteristics:

(a) The initial value of one data member depends upon the value of another data member.

(b) The dependent data member is declared first in the header file.

(c) The dependent data member is used later.

Inheritance:

14. **Redefine an inherited non-virtual member function. (inherit-redef-nvmf)**

Declaring a member function to be virtual tells the compiler that this object should always call its own version of that function. If an inherited non-virtual function is redefined, the function that will actually be called depends on how the object is referenced, because non-virtual functions are statically bound [Mey92]. For example, suppose the `rotate(degrees)` function for the `Shape` class described in Section 1.1 was not defined to be virtual, and that the `Square` class redefined it anyway. If a `Square` object is referred to by a pointer to a `Square` object, as in the example shown in Figure 5 below, the `Square` version of the `rotate(degrees)` function will be called. However, if the same object is referred to by a pointer to a `Shape` object, the `Shape` version will be called. The same result occurs with a reference as with a pointer.

```
Square square (1, 1, 5, 5, blue);

Square* ptr1 = square;
ptr1->rotate(degrees); // Calls Square::rotate(degree)


Shape* ptr2 = square;
ptr2->rotate(degrees); // Calls Shape::rotate(degrees)
```

Figure 5: **Behavior of a redefined non-virtual function**

Characteristics:

(a) An inherited, non-virtual function is redefined.

(b) A pointer or reference is used to refer to a derived object as its parent's class.

(c) The function is called via this pointer or reference.

15. **Redefine an inherited default parameter. (inherit-redef-param)**

Functions may be dynamically bound by being declared virtual, but default parameter values are always statically bound [Mey92]. These two rules are inconsistent, but were defined this way for efficiency purposes: the overhead of virtual functions was deemed acceptable; the overhead of virtual default parameter values was not.

Suppose the `Shape` class from Section 1.1 defines a function `setColor` with a parameter `color`, and the default parameter value is `red`. Then suppose `Square` redefines `setColor`, giving it the default parameter value of `green`. When the `setColor` function is called for a `Square` object, the value of the parameter will be `red` and the redefined value is ignored.

Characteristics:

(a) A default parameter is redefined for a redefined virtual function.

(b) A pointer or reference is used to refer to an object of a derived class as its parent's class.

(c) The function is called using the pointer or reference.

16. **Cast down the inheritance hierarhy. (inherit-cast-down)**

Given a pointer to a base class object, *casting down* means assuming that the object is a member of one of the derived classes and forcing (via an explicit cast) the pointer to be treated as a pointer to an object of that derived class. A failure may result if the object pointed to is not actually a member of the class it is cast to, and a function of that class is called for the object.

Characteristics:

(a) A pointer or reference is used to refer to its parent's class.

(b) The pointer or reference is cast to be one of its sibling classes.

(c) A function of the sibling's class (one that does not exist for this class) is called.

17. **Pass and return objects of a derived class by value. (inherit-slicing)**

This type of potential fault, also called the "slicing problem," occurs when an object of a derived class is passed by value into a function expecting an object of the parent class. The new object created as the local object will be a member of the parent class, and information

pertaining to the derived class does not exist in this object. A failure may occur if a function defined by the derived class is called or if information specific to the derived class is required.

This slicing problem also occurs when a function expects to return an object of a parent class, but an object of a derived class is actually returned. Again, the object returned is an object of the parent class, and any information pertaining to the derived class does not exist in this object.

Characteristics:

(a) An object of a derived class is passed by value into a function that is expecting an object of its parent's class.

(b) A function of the derived class is called for the object that is local to the function.

Or,

(a) An object of a derived class is returned from a function defined to return an object of its parent's class.

(b) A function of the derived class is called on the returned object.

18. **Duplicate in a derived class the name of a data member used in a parent class. (inherit-member-name)**

Suppose two classes are defined, one derived from the other. If the derived class duplicates the name of a data member used by any parent class, the results will probably be incorrect.

When an object is constructed, typically a constructor is run for each of its parent classes, and their parent classes, and so on, up the hierarchy to the base class(es). Since the constructor for the parent class is run before the constructor for the derived class, the constructors run in order from the base class to the most derived class. At the time that any constructor is running, the object does not "know" what type of object it "will be". At that moment, the object is a member of the same class as the constructor that is running.

Thus, for this type of potential fault, when the object's parent class constructor is running, the constructor initializes that class' data member, for example, ParentClass::my_name. When the object is fully created, the data member used will be the instance of the data member defined by the derived class, for example, ChildClass::my_name. A failure may result if the data member ChildClass::my_name is used before it is initialized.

Characteristics:

(a) In a derived class, the name of a data member used in a parent class is duplicated.

(b) The data member of the derived class is used before its value is set via a function of the derived class.

19. **Invoke a virtual function from the constructor of a parent class that will be called by a derived class. (inherit-virt-func)**

    This type of potential fault is very similar to #18. As described above, when an object is created, a constructor may be run for each of its parents, and each of their parents, all the way up the hierarchy. Since the constructor for the parent class is run before the constructor for the derived class, the constructors are run in order from the base class to the most derived class. At the time that any constructor is running, the object does not "know" what type of object it "will be". At that moment, the object is a member of the same class as the constructor that is running.

    Suppose a virtual function is called inside the constructor of BaseClass, the class from which ChildClass is derived. When a ChildClass object is constructed, the constructor of BaseClass will be run first. When the virtual function is called, the BaseClass version of this function will be called, not the ChildClass version.

    Characteristics:

    (a) A virtual function is derived in the parent class.

    (b) This function is called from a parent class constructor.

    (c) This function is redefined in the derived class.

    (d) That parent class constructor is called from the derived class constructor.

Encapsulation:

20. **Return a reference to a protected or private data member from a const member function. (encap-ret-ref-const)**

21. **Return a pointer to a protected or private data member from a const member function. (encap-ret-ptr-const)**

22. **Return a reference to a protected data member from a public function. (encap-ret-ref-prot)**

23. **Return a pointer to a protected data member from a public function. (encap-ret-ptr-prot)**

    All four of these types of potential faults return a handle (a pointer or reference) to internal data, which allows the caller to modify the state of the object directly. This is particularly serious for potential fault types 20 and 21, which are `const` functions. `Const` functions should ensure that the state of the object will not change.

    Characteristics:

    (a) A reference/pointer to a data member is returned from a (`const` for Fault types 20 and 21) member function.

    (b) The value returned is modified.

    (c) The object is used again such that the data member changed is also used.

3.1.2 Empirical Procedure

After the 23 potential fault types were identified, two programs were chosen to insert the faults into. The first was the MiStix file system [AO94], which was used as an example in the paper discussing the refinement of the category-partition method [AO94]. The original version was written in C, so it was rewritten as an object-oriented system using six small classes, including one derived class. The second program exercises a small inheritance hierarchy of string validation classes. Figure 6 shows these classes. See Appendix A for the source code for the MiStix file system and Appendix B for the source code for the validation program.

We were concerned about possible bias since one person would generate the test cases and insert the faults. The concern was that knowledge of one task might influence choices made when doing the other task. We decided that knowledge of test cases would be less likely to impact the insertion of faults than vice versa, because fault insertion is based on following a set of clearly defined rules. For this reason, we generated the test cases before inserting faults. (For the validation program, the categories and choices were complete, but the test frames and test cases were not finished before fault insertion began. Generating test frames and test case values is a purely mechanical procedure that requires no decisions on the part of the tester, so this step could not result in a bias.)

To generate the test cases, previously developed formal specifications for the MiStix file system [AO94] were used with minor modifications for input validation. Then the steps of the category-

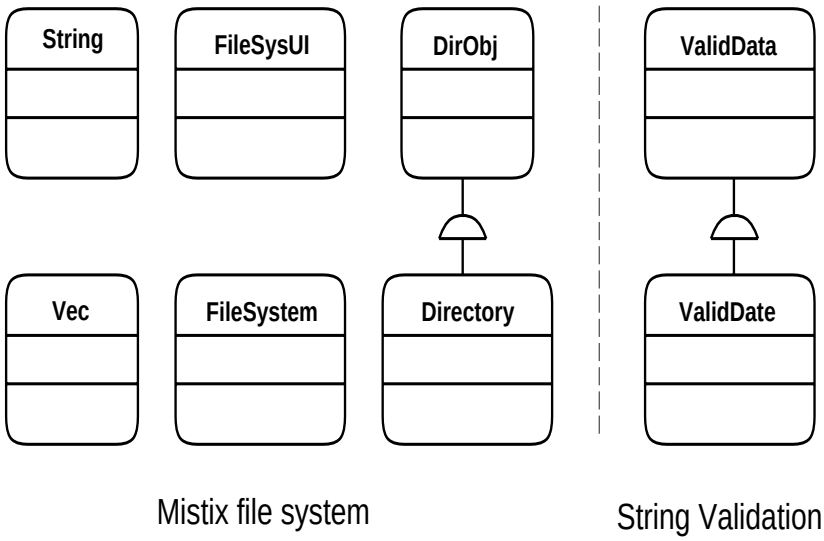

Figure 6: **Class Relationships of Test Programs**

partition method described in Section 1.4 were followed. 106 test cases were generated for the MiStix program. 31 test cases were generated for the validation program, also using the refinement to the category-partition method. The test specifications and test cases for MiStix are shown in Appendix C, and the test specifications and test cases for the string validation program are shown in Appendix D.

Faults were inserted by determining the characteristics necessary to complete the potential fault. These characteristics are shown for each type of fault in Section 3.1.1. Without all the characteristics of a fault, the program may be equivalent to the original. Knowledge of the program was used to determine these characteristics. Nineteen of the twenty-three fault types were inserted into the MiStix program, creating sixty faulty programs. Appendix E contains the source code differences between the original MiStix program and each fault-inserted program. These programs were compiled with the g++ version 2.6.0 compiler on a Sun workstation running SunOS 4.1.3.

The four remaining fault types, all inheritance faults, could not be inserted into MiStix, so they were inserted into the string validation program, creating fifteen faulty programs. These programs were compiled with the SunC++ 2.1 compiler on a Sun running SunOS 4.1.3. The source code differences between the original validation program and each fault-inserted program are in Appendix F.

Table 1 shows which program each type of fault was inserted into and how many faulty programs were created. Section 3.1.4 contains more information about how the faults were inserted.

Fault Type Number	Fault Type Identifier	Number of Faults	Program
1	mm-nf-builtin	4	MiStix
2	mm-md-builtin	5	MiStix
3	mm-nf-object	2	MiStix
4	mm-del-arr	2	MiStix
5	mm-no-del	4	MiStix
6	mm-ret-local-ref	2	MiStix
7	mm-ret-new-ref	2	MiStix
8	impl-no-cc	2	MiStix
9	impl-no-op=	1	MiStix
10	impl-nonvirt-destr	1	MiStix
11	impl-msng=-op=	6	MiStix
12	impl-dup-name	5	MiStix
13	init-dep-member	5	MiStix
14	inherit-redef-nvmf	1	MiStix
15	inherit-redef-param	2	Validation
16	inherit-cast-down	2	Validation
17	inherit-slicing	7	Validation
18	inherit-member-name	2	MiStix
19	inherit-virt-func	4	Validation
20	encap-ret-ref-const	4	MiStix
21	encap-ret-ptr-const	3	MiStix
22	encap-ret-ref-prot	5	MiStix
23	encap-ret-ptr-prot	4	MiStix
Total		75	

Table 1: **Number of Faults Inserted For Each Fault Type and Program**

Both sets of programs were compiled and the output from the compiler was examined. Then the programs were run with their respective test cases. If the faulty program crashed while running the test cases, the fault was considered to have been detected. The output of each program that did not crash was compared to the output of the original programs using the UNIX utility `diff`. (The output of the original program had been validated by hand before fault insertion began.) A difference in the outputs was considered a failure, and the fault was considered to have been detected. This is valid for these programs, since they have a single correct answer, i.e. no concurrency or output tolerance interval.

3.1.3 The Results

As shown in Table 2, of the seventy-five fault-inserted programs, fifty-five were detected using the category-partition method. This means that these programs either produced a difference in the output or crashed while running the test scripts. Twenty faults were not detected by category-partition.

One of these undetected faults could have been detected by category-partition, but was not due to an artifact of the conduct of the experiment, specifically, because of the way the test scripts were written. At the beginning of each test script, the INIT command was given. In most cases, this command is redundant, since a correctly operating program will put the file system into a valid, empty state when it begins running. In the case of this undetected fault, however, the file system was not initially put into a valid state. But because the INIT command was then run, the file system was put into a valid state, so none of the test cases detected the fault. Beginning each script with INIT was an artifact of the way the test scripts were created, and not related to the category-partition method of generating the test cases. For this reason, this fault could have been detected by category-partition.

As shown in Tables 3 and 4, the other nineteen undetected faults were all memory management faults. These types of faults are typically memory leaks, which do not affect the output of a program. We would not expect to detect memory leaks through a method of testing that analyzes output; a method that analyzes the allocation and deallocation of memory is required.

	Number of Faults	Percent of Faults
Detected	55	73.3%
Could Have	1	1.3%
Not Detected	19	25.3%

Table 2: **Results by Number and Percentage**

Category	Detected	Could Have	Not Detected
Memory Management	2	0	19
Implicit Functions	15	0	0
Initialization	4	1	0
Inheritance	18	0	0
Encapsulation	16	0	0
Totals	55	1	19

Table 3: **Results by Category**

Fault Type Number	Fault Type Number	Detected	Could Have	Not Detected
1	mm-nf-builtin	0	0	4
2	mm-md-builtin	0	0	5
3	mm-nf-object	0	0	2
4	mm-del-arr	2	0	0
5	mm-no-del	0	0	4
6	mm-ret-local-ref	0	0	2
7	mm-ret-new-ref	0	0	2
8	impl-no-cc	2	0	0
9	impl-no-op=	1	0	0
10	impl-nonvirt-destr	1	0	0
11	impl-msng=-op=	6	0	0
12	impl-dup-name	5	0	0
13	init-dep-member	4	1	0
14	inherit-redef-nvmf	1	0	0
15	inherit-redef-param	2	0	0
16	inherit-cast-down	2	0	0
17	inherit-slicing	7	0	0
18	inherit-member-name	2	0	0
19	inherit-virt-func	4	0	0
20	encap-ret-ref-const	4	0	0
21	encap-ret-ptr-const	3	0	0
22	encap-ret-ref-prot	5	0	0
23	encap-ret-ptr-prot	4	0	0
Total		75	1	19

Table 4: **Results by Fault Type**

3.1.4 Discussion

To evaluate the effectivenes of category-partition with object-oriented programs, we would like to select faults that represent small semantic changes to the programs. A semantic change is a change in meaning or interpretation of some part of a program. For example, a function that calculates a distance in miles is changed to calculate the distance in kilometers. We define the *fault size*, or the size of a semantic change, as the number of inputs for which the output of the modified program is different from the output of the original program. Large semantic changes, such as the distance function example above, would be caught on almost any input and would therefore bias the results in favor of category-partition.

A semantic change to a program is implemented by making syntactic changes to the source code.

A syntactic change is a change in the form of the source code, for example, changing a `for` loop to a `while` loop. The number of syntactic changes is not necessarily related to the size of the semantic change. A single syntactic change, such as replacing a "+" with a "*" in an initialization statement in an algorithm, represents a large semantic change, while many syntactic changes are required to change a data member from being created automatically with the object to being created by `new` in the constructor, even though this is a small semantic change. So, although the differences between the original programs and the fault-inserted programs may occasionally be extensive, the size of the semantic change is the significant factor.

In addition to making syntactic changes to implement the semantic change for each potential fault, if a program did not have all the required characteristics of the fault when the potential fault was added, the missing characteristics were also inserted. Care was taken that these additional characteristics were natural statements that could occur and that they alone would not cause a failure. For example, Fault 10-1, making a base class destructor non-virtual, is only a fault if the base class dynamically allocates memory. In order to insert this fault into the DirObj class, one of the data members needed to be dynamically allocated. Since the design of this class could have been implemented by dynamically creating this data member, a set of modifications were made to the class to do so.

In other cases, a fault might be placed in a function that was not called in the original program. For example, the fault type might be that an assignment to a particular data member is neglected in the assignment operator (Fault type 11). The assignment operator obviously must be called for this function to be detected. Since no assignment operators were used in the original program, use of assignment operators was forced where it could have occurred naturally. For example, the String assignment operator was not used in the original program, but an assignment operator that took a `char*` was. So to cause the String assignment operator to be called, a String object was made from the `char`*, then the String object was used where the `char`* had been used.

The results show that the category-partition method found all but one of the non-memory management faults. The one that it did not find actually demonstrated a fault in the way that the test scripts were written: the INIT command should not have been called. If it had not, this fault also would have been detected. This tells us that the category-partition method is effective at detecting certain types of faults for these two programs: faults involving implicit functions, initialization, inheritance and encapsulation.

The fact that only two of the memory management faults were detected shows that category-

partition was not effective at detecting these types of faults, which are typically memory leaks. As noted in Section 3.1.3, we would not expect to detect memory leaks through a method of testing that analyzes output; a method that analyzes the allocation and deallocation of memory is required. The two faults of Fault type 6, which are memory management but not memory leak faults, caused warnings to be generated by the gnu compiler, but since they caused no difference in the output of the program, they were not considered to have been detected by category-partition.

3.1.5 Measuring Fault Size

As defined in the previous section, the fault size provides a measurement of how many test cases in a test set detect a fault. Measuring the size of a fault is interesting because it gives more information about the effectiveness of a testing strategy. For example, a fault that is detected by 98% of the test cases is a relatively large fault and would be less interesting than a fault that is detected by 1% of the test cases.

At the same time, analysis of fault sizes is difficult for two primary reasons. First, we do not have a basis for comparison. We cannot say that our fault sizes are better or worse than any others. Second, we have no measurement of a "good" fault size. We want our faults to represent those made by typical programmers, but we have no measurement of "typical" faults.

The following two tables show the sizes of the faults inserted for this project. The first table shows, for each fault, the percentage of test cases that detected the fault, and which program this fault was inserted into. (MiStix had 106 test cases; the String Validation program had 31 test cases.) The second table shows a summary of these results. Faults that are not shown in the tables were not detected, i.e., the fault size was 0.

4 Conclusions

This paper presents empirical data that show that the category-partition testing technique is very effective at finding faults in object-oriented software. We conclude that existing testing techniques are effective for testing object-oriented software, and new techniques are not necessarily needed.

The research on object-oriented software testing to date has focused on two questions: *How can we use the properties of object-oriented software to reduce the effort required to test object-oriented*

Fault Number	% of Test Cases	Program
4-1	96.2%	MiStix
4-2	100.0%	MiStix
8-1	100.0%	MiStix
8-2	100.0%	MiStix
9-1	100.0%	MiStix
10-1	80.0%	MiStix
11-1	1.9%	MiStix
11-2	1.9%	MiStix
11-3	7.5%	MiStix
11-4	.9%	MiStix
11-5	2.8%	MiStix
11-6	2.8%	MiStix
12-1	7.5%	MiStix
12-2	2.8%	MiStix
12-3	.9%	MiStix
12-4	1.9%	MiStix
12-5	99.0%	MiStix
13-2	98.1%	MiStix
13-3	98.1%	MiStix
13-4	98.1%	MiStix
13-5	.9%	MiStix
14-1	88.7%	MiStix
15-1	6.5%	Validation
15-2	51.6%	Validation
16-1	45.2%	Validation
16-2	45.2%	Validation
17-1	51.6%	Validation
17-2	51.6%	Validation

programs? and *How can we effectively test object-oriented programs?* The latter question involves two issues: whether traditional techniques are effective for object-oriented software and whether new techniques need to be developed. We focused on the first of these issues.

We examined the effectiveness of the category-partition method at detecting faults in C++ programs. First, we identified 23 types of faults that are common to C++ programs and two programs to insert faults of these types into. A refinement to the category-partition method was used to generate 137 test cases for both programs and these were put into test scripts. Then faults were inserted into the program, creating 78 faulty programs. Finally, the faulty programs were run against the test scripts. A fault was considered detected if it caused the program to crash or if the output was different from the output of the original program.

Fault Number	% of Test Cases	Program
17-3	51.6%	Validation
17-4	96.8%	Validation
17-5	22.6%	Validation
17-6	51.6%	Validation
17-7	29.0%	Validation
18-1	100.0%	MiStix
18-2	88.7%	MiStix
19-1	45.2%	Validation
19-2	22.6%	Validation
19-3	22.6%	Validation
19-4	22.6%	Validation
20-1	7.5%	MiStix
20-2	.9%	MiStix
20-3	8.5%	MiStix
20-4	3.8%	MiStix
21-1	100.0%	MiStix
21-2	1.9%	MiStix
21-3	.9%	MiStix
22-1	.9%	MiStix
22-2	36.8%	MiStix
22-3	7.5%	MiStix
22-4	1.9%	MiStix
22-5	2.8%	MiStix
23-1	.9%	MiStix
23-2	1.9%	MiStix
23-3	3.8%	MiStix
23-4	2.8%	MiStix

Table 5: **A Measure of Fault Size**

The results of this study show that the category-partition method is effective for detecting certain non-memory leak types of faults in these two C++ programs. This study also shows clearly that memory management types of faults are not likely to be found using category-partition. However, memory management faults are not unique to object-oriented programs, and there are effective testing techniques available, with tools already on the market1 , to help detect them.

These results indicate that the combination of the category-partition method and a tool for detecting memory management faults may be effective for testing C++ programs in general. Since there is no evidence that traditional techniques are not effective, we may not need to develop new methods of testing object-oriented programs.

# of Faults	% of Test Cases
15	80 – 100%
14	20 – 52 %
26	.9% – 10%
20	0% (not detected)

Table 6: **Summary of Fault Size Data**

5 Future Work

This project examined one small part of the issue of how to effectively test object-oriented software. This study examined one specification-based technique with two small programs. It should be replicated with larger programs which may provide opportunities to insert faults more naturally and may provide opportunities to insert other types of faults. Similar studies should be performed using other testing techniques, both black box and white box. C++ was chosen for this project, but the programming language used should not be significant for a specification-based testing technique. The programming language may be significant for white-box testing, however.

This study implemented system-level tests. A study at the unit-level would indicate whether unit-level tests are as effective as system-level tests.

A more objective way of inserting faults into programs would be helpful. Specifying all the pieces required to complete a fault before attempting to insert the fault helps make the process more objective. There are situations, however, when it is up to the person inserting the faults to determine whether the other pieces of the fault can be put in – whether they reflect a design decision and whether they would be considered "naturally occurring."

6 Acknowledgements

We would like to thank Pi-Hui Hsiang for help preparing the Latex version of this report.

References

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–146, March 1976.

[AO94] P. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[Boo91] G. Booch. *Object-Oriented Design With Applications*. Benjamin-Cummings Publishing Co. Inc., Reading, MA, 1991.

[CE91] P. Coad and Yourdon E. *Object-Oriented Analysis*. Prentice Hall, second edition, 1991.

[CM90] T. E. Cheatham and L. Mellinger. Testing object-oriented software systems. In *1990 ACM Eighteenth Annual Computer Science Conference*, pages 161–165, February 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):481, Decmber 1985.

[DF91] R. K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 165–177, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company Inc., Reading, MA, 1990.

[Fie89] S. P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–74, April 1989.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[FW91] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 154–164, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[HM92] M. J. Harrold and J. D. McGregor. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering*, pages 68–80, Melbourne, Australia, May 1992. IEEE Computer Society.

[How76] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.

[How85] W. E. Howden. The theory and practice of function testing. *IEEE Software*, 2(5), September 1985.

[Mey92] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Publishing Company Inc., 1992.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[PK90] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal of OOP*, 2:13–19, Jan/Feb 1990.

[RW82] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *Software Engineering 6th International Conference*. IEEE Computer Society Press, 1982.

[SR90] M. D. Smith and D. J. Robson. Object-oriented programming – the problems of validation. In *Proceedings of the 1990 IEEE Conference on Software Maintenance (CSM-90)*, pages 272–281, San Diego, CA, Nov 1990.

[TR93] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM-93)*, September 1993.

[WC80] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[Wey88] E. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):676–686, June 1988.

[Whi87] L. J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, Inc, 1987.

[WSHF81] W. A. Wulf, M. Shaw, P. N. Hilfinger, and L. Flon. *Fundamental Structures of Computer Science*. Addison-Wesley Publishing Company Inc., Reading, MA, 1981.