skip me

skip me

Testing Object-Oriented Software Using the Category-Partition Method

A. Jefferson Offutt[†]
Alisa Irvine
ISSE Department, MS 4A4
George Mason University
Fairfax, VA 22030
email: ofut@isse.gmu.edu

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 and developers need to understand effective ways to test the software. Much previous work in testing OO software has focused on developing new techniques and procedures. We ask whether existing techniques can work, and present empirical data that show that the existing technique of category-partition testing can effectively find faults in object-oriented software, and new techniques may not be 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 the evidence is that traditional techniques are effective, this paper suggests that software developers will not need new testing methods when migrating to object-oriented development.*

[†] Partially supported by the National Science Foundation under grant CCR-93-11967.

1 Introduction

Testing software is one of the most common methods used to ensure that software is highly reliable. Much is known about testing traditional software, but as developers migrate to object-oriented software, developers must learn effective ways to test software in the new environment. Much of the research in testing object-oriented software has focused on developing new techniques and processes to test object-oriented software. This begs the question of whether new testing techniques are really needed, or whether existing techniques can be used. While some may think it is self-evident that traditional black-box testing techniques will work well on object-oriented software, the recent literature implies that many researchers do not. The general purpose of the research reported in this paper is to investigate whether existing testing techniques can be effectively applied to object-oriented software.

To do this, we take a case study approach to the problem, and apply software testing to object-oriented software. We choose to focus on specification-based testing, and use the category-partition method because it has been used effectively on software that was based on data abstraction and information hiding, thus it seems likely that this technique is 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 testing object-oriented software, and then discusses some of the previous research in software testing. This discussion relates the research to 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, if not, the development of new techniques. The research here attempts a partial answer to the first of these issues. Section 2 describes our approach to measuring the effectiveness of the category-partition method for testing C++ programs and presents results from a case study. Section 3 presents conclusions and Section 4 considers future work.

1.1 Software Testing

Test requirements are specific things that must be satisfied or covered; e.g., reaching statements are the requirements for statement coverage, killing mutants are the requirements for mutation, and executing DU pairs are the requirements in data flow testing. A *testing criterion* is a rule or collection of rules that impose test requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of the test requirements that are satisfied. A *testing technique* guides the tester through the testing process by including a testing criterion.

Software testing techniques are roughly divided into two categories: white box and black box [Whi87]. *White box* techniques explicitly use the structure of the program to generate test data, and are also known as structural techniques, *black box* techniques generate test data based on abstract descriptions of the software such as specifications or requirements, without using knowledge of the code or the structure of the software.

Examples of white box testing methods are path analysis [How76], data flow testing [RW85, FW88], domain testing [Whi87], and mutation testing [DLS78, Ham77]. Examples of black box testing methods are functional testing [How85], specification-based testing [GMH81, Hay86], and category-partition testing [OB88, BHO89]. This project used the category-partition method, as described in Section 2.1.2.

1.2 Testing Object-Oriented Software

As stated before, previous object-oriented software testing research has focused on two general goals: using the properties of object-oriented software to reduce the effort required to test object-oriented programs, and finding ways to effectively test object-oriented programs. The first goal follows the intuitive notion that it should be possible to use object-oriented language features such as inheritance, encapsulation and parameterized classes to reduce the effort involved in testing software. The second goal involves two problems: deciding whether traditional techniques are effective for object-oriented software and, if not, developing new techniques. Most of the research so far has been aimed at either reaching the first goal or developing new techniques, and little has been done to evaluate the effectiveness of traditional techniques.

There seems to be general agreement that effort in testing may be reduced by taking advantage of inheritance relationships where member functions are inherited unchanged from the base class [Fie89, CM90, DF91]. The extent of effort that may be saved and the best technique to use are not yet clear. For example, Perry and Kaiser [PK90] disagree with the intuitive notion that classes may be reused without re-testing in the new, derived context. They apply Weyuker's test adequacy axioms [Wey86] to object-oriented features, and show that when a well tested class is used as a superclass, if the derived classes use encapsulation, overriding of inherited methods, and multiple inheritance, the derived class will need substantial retesting.

Cheatham and Mellinger [CM90] claim that traditional white box and black box techniques are effective. Their arguments are made intuitively, without evidence. Several papers have claimed that traditional techniques (alone) are insufficient to test OO software [Fie89, SR90, TR93], but only Fiedler [Fie89] has any evidence, and that evidence is inconclusive. The results from Fiedler's study indicate that specification-based testing is not effective for testing object-oriented software, but it should be noted that the development team itself performed specification-based tests. Software developers are notoriously bad at finding defects in software they have developed [Bei90], thus this study included a strong bias against the testing.

Turner and Robson [TR93] do not reject traditional techniques for object-oriented software, but they do not believe that they are thorough enough. They are developing a technique for use in addition to traditional techniques; as yet, there is no evidence that using their technique improves testing results. Some work is being done to improve the process of testing software, principally by using the features of object-oriented software to reduce the effort involved in testing. Doong and Frankl [DF91] use algebraic specifications to derive test cases, and find that in some cases inheritance allows shortcuts to be taken. 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.

To our knowledge, no one has attempted to answer the question of whether traditional techniques work well for testing object-oriented programs. This is particularly important given the facts that there is ongoing research into finding new ways to test object-oriented software and that object-oriented software presents the potential for entirely new kinds of software faults.

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

Many techniques for testing software are known and used, thus before developing new techniques, it seems reasonable to ask whether any of these traditional techniques will effectively test object-oriented software. It is of course impossible to analytically show that all techniques or any particular technique will work for all object-oriented software, thus we restrict our attention to one technique and one object-oriented programming language. C++ was chosen as the programming language because it is one of the most widely used object-oriented programming languages and because some of its peculiarities present many potential problems [Mey92]. We chose the category-partition technique for testing because it was judged to be appropriate for object-oriented software. Since category-partition is a specification-based technique, the programming language used is not necessarily significant in the quality of the tests.

2.1 The Approach and Experimental Artifacts

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. Faults were inserted into the two programs, and they were then run with the test cases. If the program terminated abnormally (crashed) or if the output was different from the output of the original program, the fault was considered to have been detected. Our empirical study involved three kinds of artifacts: test cases, programs, and faults, as described in the following three subsections.

2.1.1 Subject programs

Two programs were chosen as subjects. The first was the MiStix file system [AO93], which was used in previous empirical research by the first author [AO94]. MiStix is a simplified version of the `Unix` file system, and accepts sequences of commands to create and manipulate files and directories. The original version was written in C, so it was rewritten as an object-oriented system in C++ using six small classes, including one derived class. The second program exercises a small inheritance hierarchy of string validation classes. Figure 1 shows these classes using Coad's notation [CE91].

2.1.2 The Category-Partition Method

The category-partition testing technique [OB88, BHO89] guides the tester to create functional test cases by decomposing functional specifications into test specifications for major functions of the software. It identifies those elements that influence the functionality and generates test cases by methodically varying the elements over all values of interest. Thus, it can be considered a black-box integration technique.

The category-partition method offers the test engineer a general procedure for creating test specifications. The test engineer's key job is to develop *categories*, which are defined to be the major characteristics of the input domain of the function under test, and to partition each category into equivalence classes of inputs called *choices*. By definition, choices in each category must be disjoint, and together the choices in each category must cover the input domain.

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 category-partition method was applied. 106 test cases were generated for the MiStix program, and 31 for the validation program. The test specifications and test cases for MiStix are given in a technical report [IO95].

2.1.3 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 six years of developing software using C++. 23 types of faults were identified: 20 from Meyers' book and 3 from experience. The fault types considered were divided into five categories: memory management, implicit functions, initialization, inheritance and encapsulation.

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 is also dereferenced

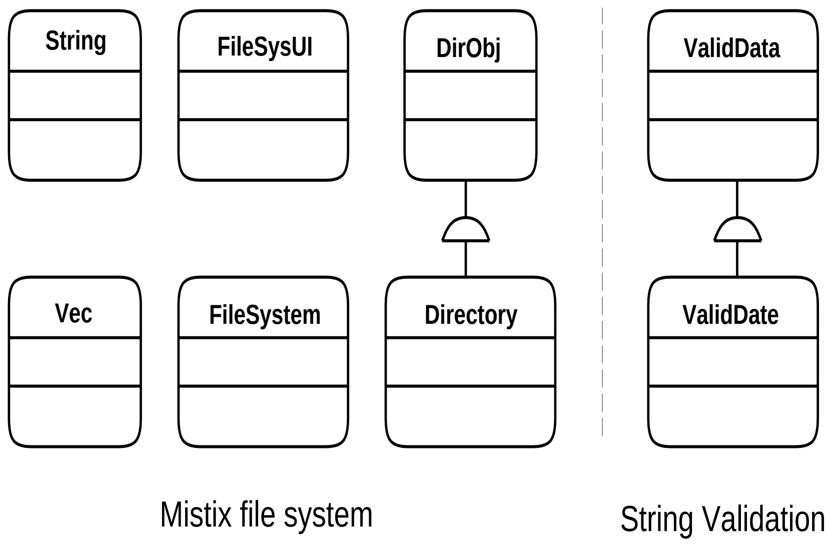

Figure 1: **Class Relationships of Test Programs**

later during execution. We were careful as we inserted faults into the programs to include all characteristics of the faults. If only part of a fault is present, it is impossible for the fault to result in a failure.

A program exhibits memory management faults when it improperly allocates or releases memory when creating or destroying objects. Two problems are common, 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. A *memory leak* occurs when memory storage is still 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. The detection of garbage requires significant execution time, however, and is not done in C++.

Implicit functions are generated automatically by C++ compilers for classes 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. Inheritance faults are related to improper design or implementation of an inheritance hierarchy. Encapsulation faults violate the principle of encapsulation, or information hiding.

The types of potential faults that we used are listed in Table 1. Although space does not allow a full description here, the potential fault types are fully explained in a technical report [IO95], and all characteristics required to create each fault are described.

2.2 Empirical Procedure

To avoid a potential bias, we generated the test cases before inserting faults. The same person generated the test cases and inserted the faults, thus a concern was that knowledge of one task might influence choices made when doing the other task. We decided that knowledge of test cases would not impact the insertion of faults, because fault insertion is based on a set of clearly defined rules. 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.

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**)
4. Allocate a single object using `new`, destroy it using `delete[]`. (**mm-del-arr**)
5. Neglect to delete a pointer data member in a destructor. (**mm-no-del**)
6. Return a reference to a local variable. (**mm-ret-local-ref**)
7. Return a reference to an object created by `new` in that function. (**mm-ret-new-ref**)

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=**)
10. Make a base class destructor non-virtual. (**impl-nonvirt-destr**)
11. Neglect an assignment to a data member within an assignment operator. (**impl-msng=-op=**)
12. Duplicate the name of a data member. (**impl-dup-name**)

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**)

Inheritance:

14. Redefine an inherited non-virtual member function. (**inherit-redef-nvmf**)
15. Redefine an inherited default parameter. (**inherit-redef-param**)
16. Cast down the inheritance hierarchy. (**inherit-cast-down**)
17. Pass and return objects of a derived class by value. (**inherit-slicing**)
18. Duplicate in a derived class the name of a data member used in a parent class. (**inherit-member-name**)
19. Invoke a virtual function from the constructor of a parent class that will be called by a derived class. (**inherit-virt-func**)

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**)

Table 1: **Description of Fault Types**

Faults were inserted by determining all characteristics necessary to complete the fault. Knowledge of the program was used to determine these characteristics. 19 of the 23 fault types were applied to create 60 faults for MiStix. The faults were applied in as many places as it was feasible. The faults are shown as source code differences between the original and the faulty programs in the technical report [IO95]. The 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 15 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 and the faulty programs are also in the technical report [IO95]. Table 2 shows which program each type of fault was inserted into and how many faulty programs were created. For example, four faults of type 1, **mm-nf-builtin**, were inserted into MiStix.

To ease the process of determining which faults were revealed, each fault was created as a separate program. Both sets of programs were compiled and the outputs from the compiler were examined to ensure that the faulty programs compiled cleanly. 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 finished execution was compared to the output of the original programs using the UNIX utility `diff`. (The outputs 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., they have no concurrency or output tolerance intervals.

2.3 Results and Discussion

As shown in Table 3, 55 of the 75 fault-inserted programs 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 testing.

One of these undetected faults could have been detected by category-partition, but was not because of 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, an INIT command was given. In most cases, this command should have been redundant, since a correctly operating program will put the file system into a valid, empty state when it begins running. The undetected fault, however did not initially put the file system into a valid state. But because the INIT command was always used, the file system was put into a valid state, so none of the test cases detected the fault.

As shown in Table 4, the other 19 undetected faults were all memory management faults. These types of faults typically cause memory leaks, which do not affect the output of a program. We would not expect to detect memory leaks by testing the functional behavior of the program; analysis of the allocation and deallocation of memory is required. Although we might have predicted that memory management faults would not be detected, experience on these matters is often wrong.

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, ini-

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

tialization, 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. As noted previously, 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.

2.4 Measuring Fault Size

To evaluate the effectiveness of category-partition with object-oriented programs, faults should represent small semantic changes to the programs. We define a semantic change in terms of the input domain. A *semantic change* is a change such that the original and modified programs behave differently on some subset of the input domain. This will usually cause a change in the 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, to be 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 dis-

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

Table 3: **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 4: **Results by Category**

tance function example above, would be caught on almost any input. Therefore, an experiment that uses many large faults are biased in favor of the testing technique used.

A semantic change to a program is implemented by making syntactic changes to the source code. A *syntactic change* is a change in 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 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 usually a small semantic change. So, although the differences between the original and the faulty programs may occasionally be extensive, the size of the semantic change is the significant factor.

The number of test cases in a given test set that detects a fault can provide an approximation of the fault size. Measuring the size of a fault 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.

Unfortunately, analysis of fault sizes is difficult. We do not have a basis for comparison – since no other researchers have provided data on fault sizes, we do not know what typical fault sizes are. We cannot say that our fault sizes are better or worse than any others. Also, 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 the size of "typical" faults.

Table 5 summarizes the percentage of test cases that detected the faults. The table shows the faults divided into four ranges, e.g., 15 faults were detected by between 80 and 100% of the test cases. Although, as stated, we are not sure whether these are typical or not, it seems encouraging that approximately a third of the faults were detected 10% of the time or less. It is also interesting that the detected faults fall into three discernable groups, one small, one medium, and one large.

3 Conclusions

This paper presents empirical data that show that the category-partition testing technique can be effective at finding faults in object-oriented software. We see no evidence that existing test-

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

Table 5: **Summary of Fault Size Data**

ing techniques are ineffective for testing object-oriented software, and conclude that new techniques may not be 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 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. 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.

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 market, 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.

4 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 that may provide more opportunities to naturally insert faults and may provide opportunities to insert other types of faults [Rin87]. 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.

References

[AO93] P. Ammann and A. J. Offutt. Functional and test specifications for the

MiStix file system. Technical report ISSE-TR-93-100, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1993.

[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.

[BHO89] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

[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.

[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.

[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.

[GMH81] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[Hay86] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.

[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.

[IO95] A. Irvine and A. J. Offutt. The effectiveness of category-partition testing of object-oriented software. Technical report ISSE-TR-95-102, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1995.

[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.

[Rin87] D. C. Rine. A common error in the object structure of object-oriented design methods. *ACM SIGSOFT Notes*, 12(4):42–44, October 1987.

[RW85] S. Rapps and W. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

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

[TR93] C. D. Turner and D. J. Robson. The testing of object-oriented programs. Technical report TR-13/92, University of Durham, Computer Science Division, January 1993.

[Wey86] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12:1128–1138, December 1986.

[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.

Category-partition Method. â€¢ Key idea. â€“ Method for creating test suites â€“ Role of test engineer. â€¢ Analyze the system specification â€¢ Write a series of formal test specifications. â€“ Automatic generator. â€¢ Produces test frames. Steps. â€¢ Decompose the functional specification into functional units. â€“ Characteristics of functional units. â€¢ They can be tested independently â€¢ Examples. â€“ A top-level user command â€“ Or a function. â€¢ Decomposition may require several stages â€¢ Similar to high-level decomposition done. by software designers. â€“ May be reused, although independent decomposition is recomm