AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Software and Systems Research Center
Technical Report

Object-Oriented programs and Testing

Dewayne E. Perry
*Gail E. Kaiser**

appears in
The Journal Of Object Oriented Programming
January/February 1990

* *Columbia University, Department of Computer Science, NY NY 10027*

1. Introduction

Brooks, in his paper ‘‘No Silver Bullet: Essence and Accidents of Software Engineering’’ [3], states:

Many students of the art hold out more hope for object-oriented programming than for any of the other technical fads of the day. I am among them.

We are among them as well. However, we have uncovered a flaw in the general wisdom about object-oriented languages — that ‘‘proven’’ (that is, well-understood, well-tested and well-used) classes can be reused as superclasses without retesting the inherited code. On the contrary, inherited methods must be retested in most contexts of reuse in order to meet the standards of adequate testing. In this paper, we prove this result by applying test adequacy axioms to certain major features of object-oriented languages — in particular, encapsulation in classes, overriding of inherited methods, and multiple inheritance pose various difficulties for adequately testing a program. Note that our results do not indicate that there is a flaw in the general wisdom that classes promote reuse (which they in fact do), but that some of the attendant assumptions about reuse are mistaken (that is, those concerning testing)

Our past work in object-oriented languages has been concerned with multiple inheritance and issues of granularity as they support reuse [10,11]. Independently, we have developed several technologies for change management in large systems [12,14,20] and recently have been investigating the problems of testing as a component of the change process [13], especially the issues of integration and regression testing. When we began to apply our testing approach to object-oriented programs, we expected that retesting object-oriented programs after changes would be easier than retesting equivalent programs written in conventional languages. Our results, however, have brought this thesis into doubt. Testing object-oriented programs may still turn out to be easier than testing conventional-language programs, but there are certain pitfalls that must be avoided.

First we explain the concepts of specification- and program-based testing, and describe criteria for *adequate testing*. Next, we list a set of axioms for test data adequacy developed in the testing community for program-based testing. We then apply the adequacy axioms to three features common to many object-oriented programming languages, and show why the axioms may require inherited code to be retested.

2. Testing

By definition, a program is deemed to be *adequately tested* if it has been covered according to the selected criteria. The principle choice is between two divergent forms of test case coverage reported by Howden [9]: specification-based and program-based testing.

Specification-based (or ‘‘black-box’’) testing is what most programmers have in mind when they set out to test their programs. The goal is to determine whether the program meets its functional and non-functional (for example, performance) specifications. The current state of the practice is informal specification, and thus informal determination of coverage of the specification is the norm. For example, tests can be cross-referenced with portions of the design document [19], and a test management tool can make sure that all parts of the design document are covered. Test adequacy determination has been formalized for only a few special cases of specification-based testing — most notably, mathematical subroutines [23].

In contrast to specification-based testing, *program-based* (or ‘‘white-box’’) testing implies inspection of the source code of the program and selection of test cases that together cover the program, as opposed to its specification. Various criteria have been proposed for determining whether the program has been covered — for example, whether all statements, branches, control flow paths or data flow paths have been executed. In practice, some intermediate measure such as essential branch coverage [4] or feasible data flow path coverage [5] is most likely to be used, since the number of possibilities might otherwise be infinite or at least infeasibly large. The rationale here is that we should not be confident about the correctness of a program if (reachable) parts of it have never been executed.

The two approaches are orthogonal and complimentary. Specification-based testing is weak with respect to formal adequacy criteria, while program-based testing has been extensively studied [6]. On the one hand, specification-based testing tells us how well it meets the specification, but tells us nothing about what part of the program is executed to meet each part of the specification. On the other hand, program-based testing tells us nothing about whether the program meets its intended functionality. Thus, if both approaches are used, program-based testing provides a level of confidence derived from the adequacy criteria that the program has been well tested whereas specification-based testing determines whether in fact the program does what it is supposed to do.

3. Axioms of Test Data Adequacy

Weyuker in ‘‘Axiomatizing Software Test Data Adequacy’’ [29] developed a general axiomatic theory of test data adequacy and considers various adequacy criteria in the light of these axioms. Recently, in ‘‘The Evaluation of Program-Based Software Test Data Adequacy Criteria’’ [30], Weyuker revises and expands the original set of eight axioms to eleven. The goal of the first paper was to demonstrate that the original axioms are useful in exposing weaknesses in several well-known program-based adequacy criteria. The point of the second paper is to demonstrate the *insufficiency* of the current set of axioms, that is, there are adequacy criteria that meet all eleven axioms but clearly are irrelevant to detecting errors in programs. The contribution of our paper is that, by applying these axioms to object-oriented programming, we expose weaknesses in the common intuition that programs using inherited code require less testing than those written using other paradigms.

The first four axioms state:

- **Applicability**. *For every program, there exists an adequate test set.*
- **Non-Exhaustive Applicability**. *There is a program P and test set T such that P is adequately tested by T, and T is not an exhaustive test set.*
- **Monotonicity**. *If T is adequate for P, and T is a subset of T’ then T’ is adequate for P.*
- **Inadequate Empty Set**. *The empty set is not an adequate test set for any program.*

These (intuitively obvious) axioms apply to all programs independent of which programming language or paradigm is used for implementation, and apply equally to program-based and specification-based testing.

Weyuker’s three new axioms are also intuitively obvious.

- **Renaming**. *Let P be a renaming of Q; then T is adequate for P if and only if T is adequate for Q.*

- **Complexity**. *For every n, there is a program P, such that P is adequately tested by a size n test set, but not by any size n-1 test set.*
- **Statement Coverage**. *If T is adequate for P, then T causes every executable statement of P to be executed.*

A program P is a *renaming* of Q if P is identical to Q except that all instances of an identifier x of Q have been replaced in P by an identifier y, where y does not appear in Q, or if there is a set of such renamed identifiers. The first two axioms are applicable to both forms of testing; the third applies only to program-based testing. The concepts of renaming, size of test set, and statement depend on the language paradigm, but this is outside the scope of this article.

4. Antiextensionality, General Multiple Change, Antidecomposition, and Anticomposition Axioms

We are interested in the four remaining (not so obvious) axioms: the antiextensionality, general multiple change, antidecomposition and anticomposition axioms. These axioms are concerned with testing various parts of a program in relationship to the whole and *vice versa*, and certain of them apply only to program-based and not to specification-based adequacy criteria. They are, in some sense, negative axioms in that they expose inadequacy rather than guarantee adequacy.

Antiextensionality. If two programs compute the same function (that is, they are *semantically close*), a test set adequate for one is not necessarily adequate for the other.

> *There are programs P and Q such that P ≡ Q,* [test set] *T is adequate for P, but T is not adequate for Q.*

This is probably the most surprising of the axioms, partly because our intuition of what it means to adequately test a program is rooted in specification-based testing. In specification-based testing, adequate testing is a function of covering the specification. Since equivalent programs have, by definition, the same specification [22], any test set that is adequate for one must be adequate for the other. However, in program-based testing, adequate testing is a function of covering the source code. Since equivalent programs may have radically different implementations, there is no reason to expect a test set that, for example, executes all the statements of one implementation will execute all the statements of another implementation.

General Multiple Change. When two programs are syntactically similar (that is, they have the *same shape*), they usually require different test sets.

> *There are programs P and Q which are the same shape, and a test set T such that T is adequate for P, but T is not adequate for Q.*

Weyuker states: ‘‘Two programs are of the *same shape* if one can be transformed into the other by applying the following rules any number of times: (a) Replace relational operator r1 in a predicate with relational operator r2. (b) Replace constant c1 in a predicate or assignment statement with constant c2. (c) Replace arithmetic operator a1 in an assignment statement with arithmetic operator a2.’’ Since an adequate test set for program-based testing may be selected, for example, to force execution of both branches of each conditional statement, new relational operators and/or constants in the predicates may require a different test set to maintain branch coverage. Although this axiom is clearly concerned with the implementation, not the specification, of a program, we could postulate a similar axiom about the syntactic similarity of specifications, as opposed to source code.

Antidecomposition. Testing a program component in the context of an enclosing program may be adequate with respect to that enclosing program but not necessarily adequate for other uses of the component.

> *There exists a program P and component Q such that T is adequate for P, T’ is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T’ is not adequate for Q.*

This axiom characterizes a property of adequacy as well as an interesting property of testing — that is, a program can be adequately tested even though it contains unreachable code. But the unreachable code remains untested, adequately or otherwise. The degenerate example is that in which Q is unreachable in P and T’ is the null set. By the Inadequate Empty Set axiom of the previous section, T’ cannot be adequate for Q. In the more typical case, some part of Q is not reachable in P but is reachable in other contexts; hence, T’ will not adequately test Q. While this axiom is written in program-based terms, it is equally applicable to specification-based testing. In particular, the enclosing program P may not utilize all the functionality defined by the specification of Q and thus could not possibly test Q adequately.

Anticomposition. Adequately testing each individual program component in isolation does not necessarily suffice to adequately test the entire program. Composing two program components results in interactions that cannot arise in isolation.

> *There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q.* [P;Q is the composition of P and Q.]

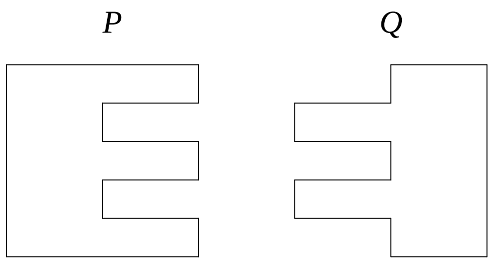

Figure 1

This axiom is counter-intuitive if we limit our thinking to *sequential* composition of P and Q. Consider instead the composition illustrated in figure 1, which can be interpreted as either P calls Q multiple times or P and Q are mutually recursive. In either case, one has the opportunity to modify the context seen by the other in a more complex manner than could be done using stubs during testing of individual components in isolation.

If the composition of P and Q is in fact sequential, then the axiom is still true — just less useful. The proof is by a simple combinatorics argument: If p is the set of paths through P and q is the set of paths through Q, then the set of paths through P;Q may be as large as $p \times q$, depending on the form of composition and on reachability as considered by the previous axiom. However, T applied to P;Q generates at most p paths. A larger test set may be needed to induce the full set of paths. This is an issue for specification-based as well as program-based testing when the specification captures only what the program is supposed to do, not including what it is not supposed to do.

5. Encapsulation in Classes

In this and the following two sections, we consider only abstractions of encapsulation, overriding of inherited methods and multiple inheritance, respectively, rather than concern ourselves with the details of specific object-oriented languages, such as Smalltalk-80 [7], Flavors [18], CommonLoops [1] and C++ [28].

Encapsulation is a technique for enforcing information hiding, where the interface and implementation of a program unit are syntactically separated. This enables the programmer to hide design decisions within the implementation, and to narrow the possible interdependencies with other components by means of the interface. Encapsulation encourages program modularity, isolates separately developed program units, and restricts the implications of changes. In particular, if a programmer changes the implementation of a unit, leaving the interface the same, other units should be unaffected by those changes. Our initial intuition, grounded in specification-based testing, is that we should be able to limit testing to just the modified unit. However, the **anticomposition** axiom reminds us of the necessity of retesting every dependent unit as well, because a program that has been adequately tested in isolation may not be adequately tested in combination. This means that integration testing is always necessary in addition to unit testing, regardless of the programming language paradigm.

Fortunately, one ramification of encapsulation for testing is that the dependencies tend to be explicit and obvious. If a programmer changes only the implementation of a unit, he need only retest that unit and any units that explicitly depend on it (call it, use its global variables, etc), as opposed to the entire program. Similarly, if the programmer adds a new unit, he need only test that unit and those existing units that have been modified to use it (plus unmodified existing units that previously used a different unit that is now masked due to a naming conflict).

One would assume that the classes of object-oriented languages would exhibit this behavior, so that it would be both necessary and sufficient to retest those classes explicitly dependent on a changed class as well as the modified class itself. We would expect that, when a superclass is modified, it would be necessary to retest all its subclasses since they depend on it in the sense that they inherit its methods. What we don’t expect is the result of the **antidecomposition** axiom — that, when we add a new subclass (or modify an existing subclass), we must retest the methods inherited from each of its ancestor superclasses. The use of subclasses adds this unexpected form of dependency because it provides a new context for the inherited components — that is, the dependency is in both directions where we thought it was only in the one direction.

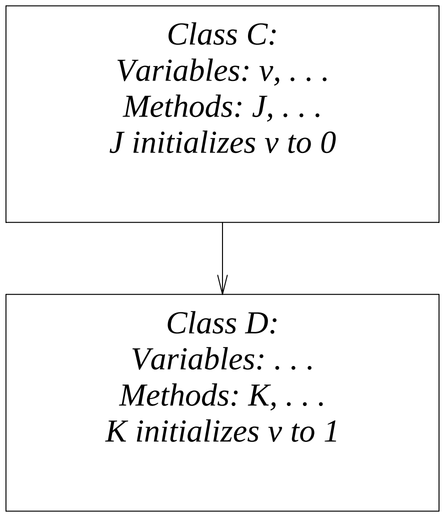

Figure 2

For example, consider a class C with method J; we have adequately tested J with respect to C. We now create a new class D as a subclass of C; D does not replace J but inherits it from C. According to the **antidecomposition** axiom, it is necessary to retest J in the context of class D. There may be new errors when in the context of D, with its enlarged set of methods and instance variables — and perhaps subtly different local *meanings* for instance variables inherited from C. The bug illustrated in figure 2 (the conflicting assumptions about instance variable v) would not be detected without retesting J in the context of D.

In order to make this example more concrete, consider C to the class WindowManager, D to be the class SunWindowManager, J is the method InitializeScreen, and K is SetScreenBackground. J initializes to a blank screen, while K puts a digitized picture in the background. There are obvious problems if K is invoked first and then J, and vice versa.

There is one case where adding a new subclass does not require retesting the methods inherited from the superclass in order to meet the adequacy axioms. This is when the new subclass is a pure extension of the superclass, that is, it adds new instance variables and new methods and there are no interactions in either direction between the new instance variables and methods and any inherited instance variables and methods.

At least one object-oriented language has solved this problem in the general case, by prohibiting unexpected dependencies: CommonObjects [25,26] removes all implicit inheritance — that is, inherited methods must be explicitly invoked. This, in effect, inserts ‘‘firewalls’’ between each superclass and its subclasses, in the same sense that encapsulation inserts firewalls between a class and its clients.

6. Overriding of Methods

Almost all object-oriented languages permit a subclass to replace an inherited method with a locally defined method with the same name, although some support a subtyping hierarchy that restricts the method to have the same specification [24]. In either case, it is obvious that the overriding subclass has to be retested. What is not so obvious is that a different test set is often needed. This is expressed by the **antiextensionality** axiom: although the two methods compute semantically close functions, a test set adequate for one is not necessarily adequate for the other.

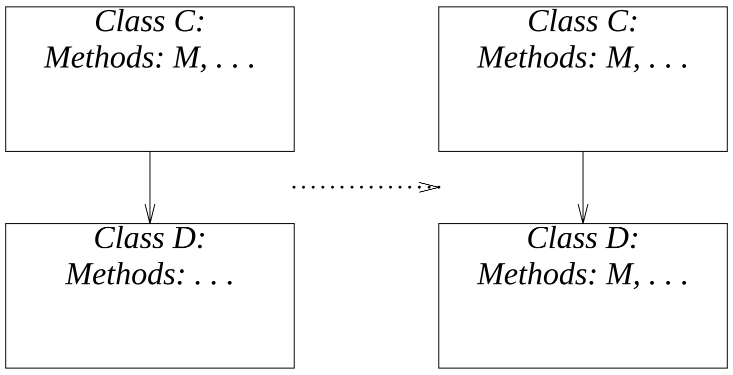

Figure 3

For example, consider figure 3 where class C has subclass D, and method M is defined in C but not in D. Say there exists an object O that is an instance of class D, which receives a message containing the method selector M; M applied to O has already been adequately tested. Now we change class D to add its own method M, which is similar to C.M (by ‘‘C.M’’, we mean the method M from superclass C). Obviously, we need to retest class D. Intuitively we would expect that the old test data would be adequate, but the **antiextensionality** axiom reminds us that it may not be adequate. Thus, we may have to develop new test cases for two reasons. First, remember that program-based testing considers the details of the program formulation, attempting to cover, for example, each statement or branch. The test data would necessarily be at least slightly different for C.M and D.M if the formulation in terms of statements and branches were different; the test data would probably be very different if C.M and D.M used different algorithms. Second, it is very likely that the underlying motivation for overriding a method affects not only the internal structure of the overriding method but its external behavior as well — that is, it changes the functional specification. Hence, in addition to test cases to exercise the different structure of the method, we need test cases to test the different specification of the that method.

More concretely, consider C to be the class WindowManager, D to be the class SunWindowManager, C.M to be the method RefreshDisplay that rewrites an entire bitmapped screen, and D.M to be the method RefreshDisplay that repaints only the ‘‘damaged’’ part of a bitmapped screen. In this case, the specifications as well as the implementations of the two methods might be different, in which case different test sets would be required for specification-based as well as program-based testing.

In the previous section, we treated the two-way dependency between classes and superclasses and explained how the **antidecomposition** axiom requires testing of inherited methods in each inheriting context as well as the defining context. What we did not discuss there was the application of the **antiextensionality** axiom to this additional testing: different test sets may be needed at every point in the ancestor chain between the class defining the overriding method and its ancestor class defining the overridden method.

In figure 4, class C has subclass D, which in turn has subclass E; C has methods M and N; D has method M, which uses method N (from C); class E does not have method M but does have method N (overriding the N inherited from C). The **antiextensionality** axiom reminds us that we need different test data for M with respect to each of the classes C, D, and E. This is obvious with respect to instances of C and D, since they invoke distinct methods M in response to the message M; even if these methods are semantically close, test data adequate for one may not be adequate for the other. This is less obvious with respect to D and E, since they invoke the identical method M. But when we consider that M calls C.N for D whereas it calls E.N for E, it becomes clear that different test sets are required since the formulation and algorithms used by C.N and E.N are likely to be different in functionality as well as structure.

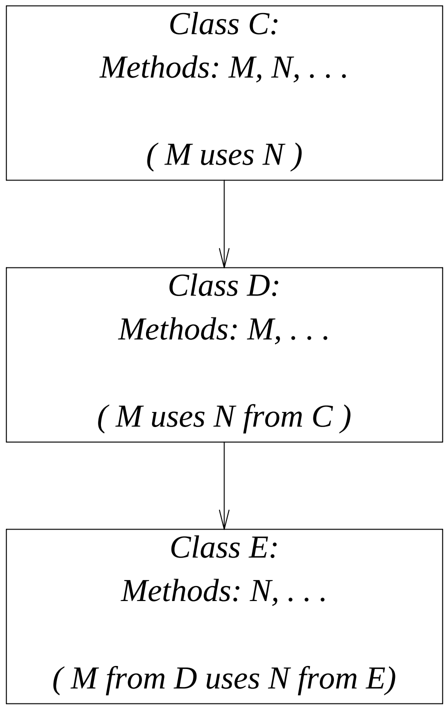

Figure 4

Again, more concretely, let class C be WindowManager where method M is RefreshDisplay and method N is DrawCharacter, using bitmapped fonts; let class D be SunWindowManager where method M is D’s replacement for the method RefreshDisplay; and let class E be NeWS where method N is E’s replacement for the method DrawCharacter, using Postscript fonts.

7. Multiple Inheritance

Some, but not all, object-oriented languages support multiple inheritance [2], where each class may have an arbitrary number of superclasses. The so-called ‘‘multiple inheritance problem’’ arises when the same component may be inherited along different ancestor paths. Solutions to this problem typically define a precedence ordering, which linearizes the set of ancestors so that there is a unique selection (or a unique ordering if the semantics of the language are such that all conflicting inherited methods must be invoked) [27]. These solutions, unfortunately, cause very small syntactic changes to have very large semantic consequences. Fortunately, the **general multiple change** axiom reminds us that programs that are syntactically similar usually require different test sets.

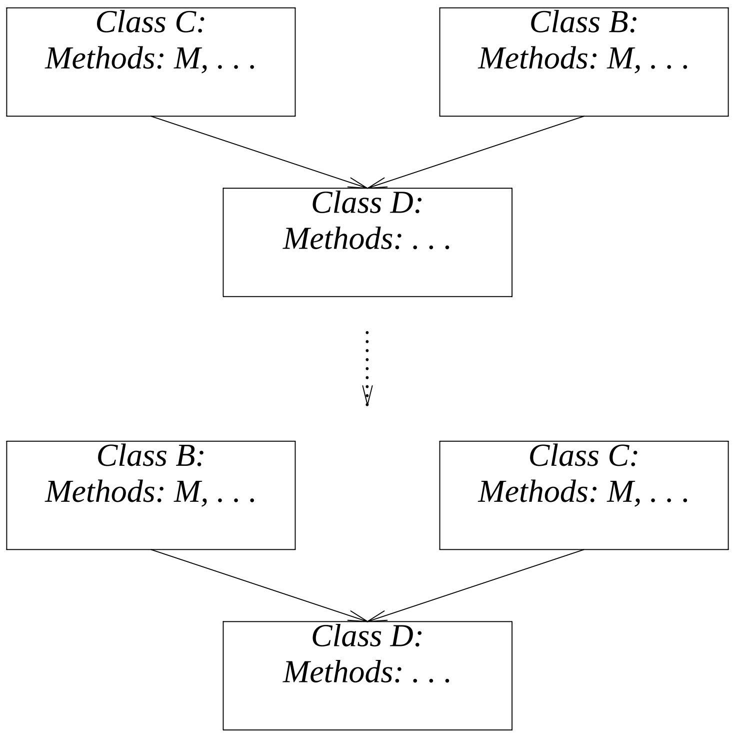

Figure 5

In figure 5, class D lists superclasses C and B, in that order, and the language imposes the precedence ordering C, B. Method M is defined by both C and B but not by D. Class D is then changed so that the ordering of the superclasses is B and C (meaning that the precedence ordering is B, C). Not only must class D be retested, since it now uses B.M rather than C.M, but most likely a different set of tests must be used. Since C and B are independent, and perhaps developed separately, there is no reason that B.M would be either syntactically or semantically similar to C.M — and even if it were, the **antiextensionality** and **general multiple change** axioms remind us that even then different test sets may be necessary.

As a concrete realization of this example, let class C be TextWindowManager where method M is RefreshDisplay (that repaints the window from a text description), let class B be GraphicsWindowManager where method M is RefreshDisplay (that repaints the window from a bit-mapped representation), and let class D be SunWindowManager.

The example in figure 6 shows the inherent compounding effects of multiple inheritance.

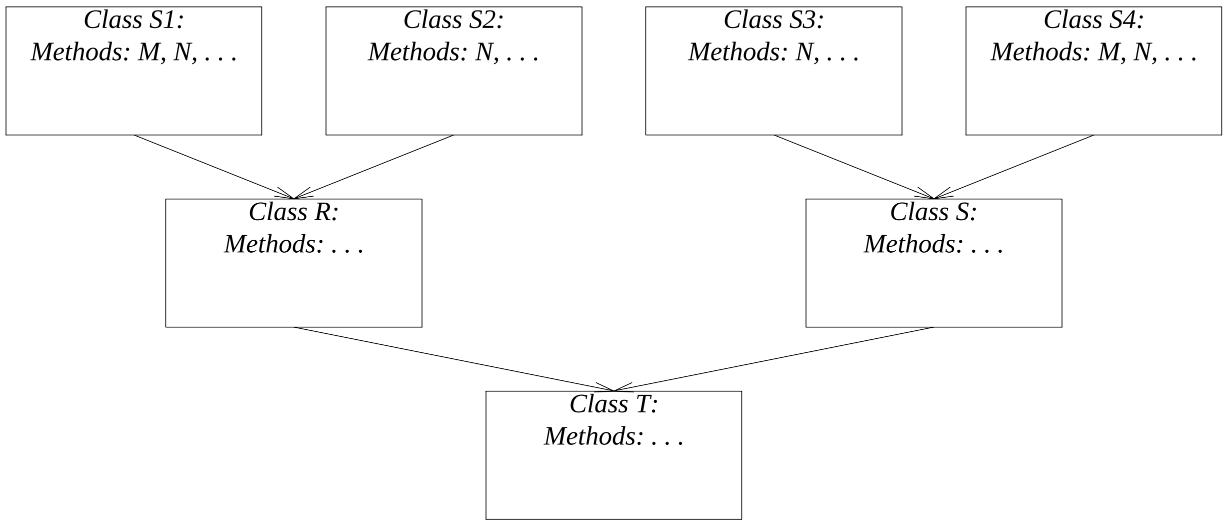

Figure 6

This implication of the **general multiple change** axiom is probably the most significant result of applying the test data adequacy axioms to object-oriented languages, but also the least surprising to the object-oriented languages community. Multiple inheritance is already widely recognized as both a blessing and a curse [15,16,17].

8. Conclusions

Inheritance is one of the primary strengths of object-oriented programming. However, it is precisely because of inheritance that we find problems arising with respect to testing.

- Encapsulation together with inheritance, which intuitively ought to bring a reduction in testing problems, compounds them instead.
- Where non-inheritance languages make the effects of changes explicit, inheritance languages tend to make these effects implicit and dependent on the various underlying, and complicated, inheritance models.

Brooks concludes his section on object-oriented programming:

> *Nevertheless, such advances can do no more than to remove all the accidental difficulties from the expression of the design. The complexity of the design itself is essential, and such attacks make no change whatever in it. An order-of-magnitude gain can be made by object-oriented programming only if the unnecessary type-specification underbrush still in our programming language is itself nine-tenths of the*

work involved in designing a program product. I doubt it.

While object-oriented programming clears away much of the accidental underbrush of design, we have noted ways in which it adds to the accidental underbrush of change management and testing. We conclude that there is a pressing need for research on testing of object-oriented languages. We have begun work on this in the context of a data-oriented debugger for concurrent object-oriented languages [8] and in the context of semantic analysis (applying the approach of Inscape [21] to C++).

Acknowledgements

Jim Coplien, Narain Gehani, Jim Krist, and Alex Wolf provided useful criticism and suggestions on earlier versions of this paper.

References

[1] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. ‘‘CommonLoops: Merging Lisp and Object-Oriented Programming’’, *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. pp 17-29.

[2] Alan Borning and Daniel Ingalls. ‘‘Multiple Inheritance in Smalltalk-80’’, *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh PA, 1982. pp 234-237.

[3] Frederick P. Brooks, Jr. ‘‘No Silver Bullet: Essence and Accidents of Software Engineering’’, *Computer* 20:4 (April 1987). pp 10-20.

[4] Takeshi Chusho. ‘‘Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing’’, *IEEE Transactions on Software Engineering* SE-13:5 (May 1987). pp 509-517.

[5] Phyllis G. Frankel and Elaine J. Weyuker. ‘‘Data Flow Testing in the Presence of Unexecutable Paths’’, in *Workshop on Software Testing*, Banff, Canada, July 1987. pp 4-13.

[6] David Gelperin and Bill Hetzel. ‘‘The Growth of Software Testing’’, *Communications of the ACM* 31:6 (June 1988). pp 687-695.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, Reading MA: Addison-Wesley, 1983.

[8] Wenwey Hseush and Gail E. Kaiser. ‘‘Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language’’, *Proceedings of the ACM SIGPlan/SIGOps Workshop on Parallel and Distributed Debugging*, Madison WI, May 1988. pp. 236-246.

[9] William Howden. *Software Engineering and Technology: Functional Program Testing and Analysis*, New York: McGraw-Hill Book Co., 1987.

[10] Gail E. Kaiser and David Garlan. ‘‘Melding Software Systems from Reusable Building Blocks’’, *IEEE Software*, July 1987. pp 17-24.

[11] Gail E. Kaiser and David Garlan. ‘‘MELDing Data Flow and Object-Oriented Programming’’, *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, Kissimmee FL, October 1987. *SIGPlan Notices* 22:12 (December 1987). pp 254-267.

[12] Gail E. Kaiser and Dewayne E. Perry. ‘‘Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution’’, *Proceedings of the Conference on Software Maintenance*, Austin TX, September 1987". pp 108-114.

[13] Gail E. Kaiser and Dewayne E. Perry. ‘‘INFUSE: Integration Testing with Crowd Control’’. Technical Report. Computing Systems Research Laboratory, AT&T Bell Laboratories, January 1988.

[14] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef. ‘‘Multiuser, Distributed Language-Based Environments’’, *IEEE Software*, November 1987. pp 58-67.

[15] Norman Meyrowitz. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. Special Issue of *SIGPlan Notices*, 21:11 (November 1986).

[16] Norman Meyrowitz. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Orlando FL, October 1987. Special Issue of *SIGPlan Notices*, 22:12 (December 1987).

[17] Norman Meyrowitz. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, San Diego CA, September 1988. Special Issue of *SIGPlan Notices*, 23:11 (November 1988).

[18] David A. Moon. ‘‘Object-Oriented Programming with *Flavors*’’, in *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. pp 1-8.

[19] Thomas J. Ostrand, Ron Sigal, and Elaine Weyuker. ‘‘Design for a Tool to Manage Specification-Based Testing’’, in *Workshop on Software Testing*, Banff, Canada, July 1987. pp 41-50.

[20] Dewayne E. Perry and Gail E. Kaiser. ‘‘Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems’’, *Proceedings of the ACM Fifteenth Annual Computer Science Conference*, St. Louis MO, February 1987. pp 292-299.

[21] Dewayne E. Perry. ‘‘Software Interconnection Models’’, *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, April 1987. pp 61-69.

[22] Dewayne E. Perry. ‘‘Version Control in the Inscape Environment’’, *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, April 1987. pp 142-149.

[23] Robert P. Roe and John H. Rowland. ‘‘Some Theory Concerning Certification of Mathematical Subroutines by Black Box Testing’’, *IEEE Transactions on Software Engineering* SE-13:7 (July 1987). pp 761-766.

[24] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. ‘‘An Introduction to Trellis/Owl’’, in *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland OR, September 1986. pp 1-8.

[25] Alan Snyder. ‘‘CommonObjects: An Overview’’, *Object-Oriented Programming Workshop Proceedings*, Yorktown Heights NY, June 1986. pp 19-29.

[26] Alan Snyder. ‘‘Inheritance and the Development of Encapsulated Software Components’’, *Twentieth Hawaii International Conference on System Sciences*, Kona HI, January 1987. volume II, pp. 227-238.

[27] Mark Stefik and Daniel G. Bobrow. ‘‘Object-Oriented Programming: Themes and Variations’’, *The AI Magazine* (Winter 1985). pp 40-62.

[28] Bjarne Stroustrup. *The C++ Programming Language*. Reading MA: Addison-Wesley, 1986.

[29] Elaine J. Weyuker. ‘‘Axiomatizing Software Test Data Adequacy’’, *IEEE Transactions on Software Engineering* SE-12:12 (December 1986). pp 1128-1138.

[30] Elaine J. Weyuker. ‘‘The Evaluation of Program-Based Software Test Data Adequacy Criteria’’, *Communications of the ACM* 31:6 (June 1988). pp 668-675.