testing desktop applications [Kaner+93]. Roper's *Software Testing* is a compact, readable survey [Roper 94]. Marick's *Craft of Software Testing* offers a meticulous, detailed test design approach of special interest to C and C++ developers [Marick 95]. Beizer's *Black Box Testing* [Beizer 95] is a focused presentation of essential implementation-independent test design strategies and covers research and practice through 1994.

Friedman and Voas's *Software Assessment* [Friedman 95] provides an insightful quantitative analysis of the problems in using testing as a means to achieve high-quality software. The *Handbook of Software Reliability Engineering* [Lyu 96] is a landmark in testing and software engineering.

Several comprehensive surveys of testing approaches have been published. An early survey of verification, validation, and testing techniques appears in [Adrion+82]. Comparative surveys of testing approaches appear in [DeMillo+87, White 87, Ntafos 88]. Morell and Deimel's SEI curriculum module on testing offers a concise and readable introduction to testing research literature. The annotated bibliography is especially useful [Morell+92]. My own survey covers object-oriented testing techniques, testing strategies for abstract data types, and approaches to automated model validation published through 1994 [Binder 96i]. The survey of testing coverage models in [Zhu+97] summarizes research reports published through 1996.

Chapter 4 With the Necessary Changes: Testing and Object-oriented Software

If there are two or more ways to do something, and one of those ways can result in a catastrophe, then someone will do it.

Edward A. Murphy, Jr.

Overview

Testing is a search for bugs, so identifying bug hazards is essential for effective testing. This chapter surveys research and experience reports about bug hazards in object-oriented programming. The idea of a fault model is presented. Several code-coverage models are reviewed and some language-specific hazards are discussed. The chapter concludes with tenets and prescriptions for effective object-oriented testing.

4.1 The Dismal Science of Software Testing

Testing is a bug hunt. To find the quarry, we must know where to look. In software, this means we must examine the ways in which good languages can go wrong. The catalog of bug hazards that follows lays a foundation for effective testing. It is not a general critique of the object-oriented programming paradigm. Just as reading a medical book on various forms of disease can be

unpleasant, you may find the following discussion unpleasant. And just as reading only a book on disease would give you a warped view of health, you should not conclude that the following catalog of problems is a definitive assessment of object-oriented programming. It is not.

4.1.1 I'm Okay, You're Okay, Objects Are Okay

On a personal level, you should not conclude from reading this chapter that you have made a bad career choice or are morally depraved because you design, program, or test object-oriented code. Nor should you conclude that I view object-oriented programming as a kind of mass hysteria, its advocates as charlatans, and its practitioners as incompetent twits. Likewise, you should not conclude that I am a retro techno-crank, obsessed with trashing object-oriented programming.

A curious kind of tribalism seems to be part and parcel of the social psychology of software development. For many people, being proficient in some technical skill is not enough: one must also become a true believer, taking the view that one's technology of choice is vastly superior and that practitioners of different approaches are misguided, at best. I have seen this attitude regarding hardware, analysis and design methodologies, operating systems, programming paradigms, and, of course, programming languages. It leads some people to think that "If you ain't with us, you agin' us." I'm not against any of the languages discussed here. I am a proponent of effective testing, which means that we must consider what can go wrong. Clearly, this analysis must be based on facts, not tribalism.

It could be argued that most, if not all, of the problems described in this chapter result from developer error. That is, if a programmer fails to use a language as it was intended, then the bug hazard is the fact that human effort generates most code, one keystroke at a time.

People make mistakes in the creation of all things. But the media that we use to express thoughts and craft artifacts limit the context of the mistake. I cannot fall down the stairs in the middle of a football field. I cannot make grammatical errors in French, Russian, or Japanese when I am writing in English. I cannot apply insufficient torque to a cylinder head bolt with a hammer. I cannot incorrectly and invisibly alter the program counter to a different instruction address in most high-level programming languages. I cannot (easily) create a single module that is a rat's nest of gotos sprawling over thousands of lines of code in any object-oriented language.

However, I can easily create subclasses that are inconsistent with their superclasses, each of which will pass many possible tests. I can easily create a maze of "spaghetti inheritance" rife with lurking bugs, but which compiles and seems to work. I cannot do these things (and others described in this chapter) in procedural languages. There can be no bugs without programmers, but the kinds of bugs we can produce are directly a result of the programming language in use.

This chapter inventories reported bug hazards in object-oriented programming languages. It is an expanded and updated version of one section from my 1996 survey of object-oriented testing [Binder 96i]. The survey summarized about 150 published research and experience reports. I had no control over what the authors of these reports studied. Some subjects received much attention, others less. This unevenness necessarily resulted in lopsided representation for some subjects. Nevertheless, this information remains the best available basis for establishing fact-based fault models. Lopsidedness should not be interpreted as a critique. For example, you should not interpret the fact that several reports on Objective-C mention initialization problems as meaning that (1) only Objective-C has this bug hazard or (2) I am singling out Objective-C for criticism or ridicule.

The perfect programming language has yet to be developed, and may never be. Object-oriented languages solved problems inherent in procedural languages that led to mountains of truly ugly and buggy code. The strengths of the object-oriented paradigm have been recited elsewhere, many times. For example, the analysis given by Wilke [Wilke 93] and Meyer [Meyer 97] is cogent and comprehensive. The absence of this recitation here does not mean that the problems discussed negate these advantages, either in fact or in my opinion. Rather, I do not feel anything would be added by repeating them yet another time.

4.1.2 The Role of a Fault Model

Testing presents fundamental and fascinating conundrums. Given that we can never hope to exercise all possible inputs, paths, and states of a system under test, which should we try? When should we stop? If we must rely on testing to prevent certain kinds of failures, how can we design systems that are both testable and efficient? How can we craft a test suite that exercises enough message and state combinations to demonstrate failure-free operation but that remains small enough to be practical?

While we can be certain there are unknown bugs in any nontrivial software system, we never know exactly where they are (if we did, we wouldn't need to test). The number of places to look is infinite for practical purposes, so any rational testing strategy must be guided by a **fault model**. A fault model answers a simple question about a test technique: *Why* do the features called out by the technique warrant our effort? This operational definition identifies relationships and components of the system under test that are most likely to have faults. The answer to the question may be based on common sense, experience, suspicion, analysis, or experiment.

A **bug hazard** is a circumstance that increases the chance of a bug. Such hazards arise for many reasons and in many situations. For example, type coercion in C++ is a bug hazard because the actual conversion depends on complex rules and on declarations that may not be visible when working on a particular class. Once identified, a bug hazard provides the basis for a fault model.

Software testing strategies are effective to the extent that their fault model is a good predictor of faults. Two general fault models and corresponding testing strategies exist:

- *Conformance-directed testing* seeks to establish conformance to requirements or specifications. Tests are designed to be sufficiently representative of the essential features of the system under test. Conformance testing relies on a **nonspecific fault model**: any fault suffices to prevent conformance. The **sufficiency** of the testing model with respect to system requirements is crucial to establish conformance. Conformance-directed testing need not consider potential implementation faults in detail, but must establish that a test suite is sufficiently representative of the requirements for a system.
- *Fault-directed testing* seeks to reveal implementation faults. It is motivated by the observation that conformance can easily be demonstrated for an implementation that contains faults. Searching for faults is a practical and prudent alternative to conformance [Myers 79]. Because the combinations of input, state, output, and paths are astronomically large, efficient probing of an implementation requires a **specific fault model** to direct the search for faults.

The choice of a fault model suggests a **test strategy**. A test strategy yields a test suite when applied to a representation or implementation. Conformance-oriented techniques should be **feature sufficient**: they should, at least, exercise all specified features. Fault-oriented techniques should be **fault efficient**: they should have a high probability of revealing a fault.

A well-formed fault model should explain why relative sufficiency or efficiency obtains for its associated testing technique. Either a convincing argument or strong evidence that a particular kind of probing has a good chance of revealing a fault is needed. **Error-guessing** relies on developer knowledge and surmise to imagine how a particular implementation could go wrong [Myers 79]. **Suspicions** are common-sense inferences—for example, code written by an inexperienced programmer is more likely to be buggy [Hamlet 90]. Fault models may be extrapolated from failure analysis. For example, some studies (discussed in the next section) have shown that deep inheritance hierarchies are more likely to be buggy than shallow hierarchies. Test strategies often rely on an assumption that a particular kind of construct is more prone to error (e.g., evaluation of boundary values in decision segments). By trying all such constructs, we will therefore find all faults of this type. For example, the data flow fault model makes this kind of assumption:

> Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed [Rapps+85, 367].

A fault model may also be based on an argument (or evidence) about the kinds of errors that are likely to be made and the kind of faults they cause. For example, the functional testing fault model is based on a correlation of faults and functions:

> Studies of program faults revealed that they are often associated with embedded subfunctions, or special functional aspects of a piece of code. The studies indicated that if these subfunctions had been separately considered in the construction of test cases, then the chance of discovering faults would have risen dramatically [Howden+93, 3].

4.1.3 Fault Models for Object-oriented Programming

The object-oriented programming paradigm presents a unique blend of powerful constructs, bug hazards, and testing problems. This is an unavoidable result of the encapsulation of operations and variables into a class, the variety of ways a system of objects can be composed, and the compression of complex runtime behavior into a few simple statements. Each lower level in an inheritance hierarchy creates a new context for inherited features; correct behavior at an upper level in no way guarantees correct behavior at a lower level. Interaction

between message sequence and state is often subtle and complex. For example, consider a class composed by multiple inheritance, with six superclasses in each contributing hierarchy and many polymorphic methods. At best, the developer must spend considerable effort to ensure that all superclass methods perform correctly in the subclass context and that no undesirable interactions occur among methods. Polymorphism and dynamic binding dramatically increase the number of execution paths. Static analysis of source code to identify paths (a bedrock technique of procedural testing) is of little use. Encapsulation can create obstacles that limit the visibility of the implementation state.

The issue of reusability and the proper role of testing in object-oriented development raise further questions. Components offered for reuse should be highly reliable; extensive testing is warranted when reuse is intended. Neither inheritance nor compositional reuse, however, reduces the need for retesting.

Object-oriented technology does not in any way obviate the basic motivation for software testing. In fact, it poses some new challenges. Despite its similarity to testing procedural systems, object-oriented testing has significant differences. Although the use of object-oriented programming languages may reduce some kinds of errors, they increase the chance of others. Methods often consist of just a few lines of code, so control-flow bugs are less likely. Encapsulation prevents bugs that result from global data scoping and intermodule side effects in some procedural languages. Nevertheless, no compelling reason (let alone evidence) exists to suppose that developers of object-oriented programs are any more immune to errors than are developers writing in procedural languages. Coding errors (misspelling, misnaming, wrong syntax) are probably as likely as ever. In addition, some essential features of object-oriented languages pose new fault hazards:

- Dynamic binding and complex inheritance structures create many opportunities for faults due to unanticipated bindings or misinterpretation of correct usage.
- Interface programming errors are a leading cause of faults in procedural languages. Object-oriented programs typically have many small components and therefore more interfaces. Interface errors are more likely, other things being equal.
- Objects preserve state, but state control (the acceptable sequence of events) is typically distributed over an entire program. State control errors are likely.

As testing is a search for bugs, focusing on these bug hazards is good testing strategy. This chapter surveys research and experience reports to identify the bug hazards of the object-oriented programming paradigm.

4.2 Side Effects of the Paradigm

4.2.1 What Goes Wrong?

Although many code-based metrics have been developed for object-oriented programming, not much empirical analysis has been published about the relationship between object-oriented code and bugs. A study on reuse found that newly written code was 48.8 times more likely to be buggy than code reused without any change (Table 4.1) [Basili+96a]. This rate for new code bugs is consistent with a report on development of a C++ system for a medical electronics application. "On average, a defect was uncovered for every 150 lines of code, and correspondingly, the mean defect density exceeded 5.1 per 1000 lines" [Fiedler 89].

TABLE 4.1 Average Bugs per KLOC and Reuse [Basili+96a]

Extent of Reuse	Faults per KLOC (thousand lines of code)
Verbatim reused code	0.125
Code slightly modified	1.500
Code extensively modified	4.890
No reuse, newly written code	6.110

Studies have been conducted using the Metrics for Object-Oriented Software Engineering (MOOSE)[1] proposed by Chidamber and Kemerer [Chidamber+94]. Two studies found a strong correlation between bugs and certain code structures measured by these metrics [Basili+96b, Briand+98]:

- Classes that send relatively more messages to instance variable and message parameter objects are more likely to be buggy.

1. The MOOSE metrics are Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response For Class (RFC), Lack of Cohesion in Methods (LCOM), and Weighted Methods per Class (WMC).

- Classes with greater depth (number of superclasses) and higher specialization (number of new and overridden methods defined in a class) are more likely to be buggy.
- No significant correlation was found between classes that lack cohesion (number of pairs of methods in a class using no attribute in common) and the relative frequency of bugs.

Both studies were conducted on small C++ systems (up to 30 KLOC). The structure metrics that were good bug predictors appeared to be better bug predictors than lines of code. The later study also evaluated approximately 50 proposed variants on the original six MOOSE structure metrics and found no significant differences in predictive power [Briand+98].

In a study of a 133 KLOC C++ telecommunications application developed using the Shlaer/Mellor OOA/D, classes that "participated in inheritance structures were three times more defect prone" than those that did not [Sheppard+97].

An earlier study of two medium-sized commercial systems written in Classic Ada (an object-oriented variant of Ada 83) found that the MOOSE metrics, taken together, were better predictors of maintenance cost (including debugging) than lines of code [Li+93]. A similar relationship is noted in [Fielder 89], although complexity and size are reported there as equally good predictors: "The number of defects found seemed to be related to the composite (total) complexity of all of the class member functions and more directly to the number of noncomment source statements (NCSS) contained in the source and include files."

Summarizing data gathered as of 1996 from 150 development organizations and 600 projects using object-oriented technologies, Jones concludes:

> (1) The OO learning curve is very steep and causes many first-use errors. (2) OO analysis and design seem to have higher defect potentials than older design methods. (3) Defect removal efficiency against OO design problems seems lower than against older design methods, which is a significant observation if confirmed. (4) OO programming languages seem to have lower defect potentials than procedural programming languages. (5) Defect removal efficiency against programming errors is roughly equal [to] or somewhat better than removal efficiency against older procedural language errors [Jones 97a, 292].

The available quantitative evidence indicates that heavier usage of structures intrinsic to the object-oriented programming paradigm is a good predictor of bugs. Testing requires a more specific fault model, however. The following survey of qualitative fault models and experienced-based analysis provides the details underlying this requirement.

4.2.2 Encapsulation

Encapsulation refers to the access control mechanisms that determine visibility of names in and among lexical units. Access control hides some names, typically to prevent unnecessary client dependencies to a server's implementation (e.g., making a class interface visible but not its implementation). It makes others visible, typically to simplify access (e.g., making superclass instance variables visible to subclasses). This supports information hiding and modularity. It helps to prevent problems with global data access common to procedural languages. Visibility in a particular system is determined by language defaults and the declarations as coded.

Although encapsulation does not directly contribute to the occurrence of bugs, it can present an obstacle to testing. Testing requires accurate reporting of the concrete and abstract states of an object and the ability to set state easily. Object-oriented languages make it difficult—if not impossible—to directly set or get the concrete state. The C++ friend function was developed to solve this problem [Stroustrup 92b]. Ada 95 child packages play a similar role.

The interaction of encapsulation and testing is illustrated by a bug in an integer set class that was part of a commercial C++ class library [Hoffman+95].

```
class IntSet {
public:
   // operations on single sets
    Intset();
    ~Intset();
   IntSet& add(int);          // Add a member
   IntSet& remove(int);       // Remove a member
   IntSet& clear(int);        // Remove all members
   int     is_member(int);    // Is arg a member?
   int     extent();          // Number of elements
   int     is_empty();        // Empty or not?

   // operations on pairs of sets ...
};
```

A precondition of `add(x)` is that `x` is not already present in the set. If this condition is present and an add message is sent, the `Duplicate` exception is thrown. To test the `Duplicate` exception, `add(1)` was invoked twice. On the second call, the exception was thrown correctly, but the duplicate value was incorrectly added to the encapsulated implementation of the set. A subsequent

message, `remove(1)`, was accepted without an exception. No further testing was done and the bug escaped detection. The bug was found (and fixed) when it was noticed that two `remove(x)` messages were required before `is_member(x)` would return `false` after two `add(x)` messages.

Chapters 17, Assertions, shows how to develop test code that is part of a class's implementation. This strategy can overcome obstacles to observing the state of an object. Chapter 19, Test Harness Design, presents several test automation patterns to deal with the obstacles to controllability that result from encapsulation.

4.2.3 Inheritance

Inheritance is essential in the object-oriented programming paradigm [Wegner 87, Stroustrup 88]. It supports reusability by allowing shared and different elements of a common entity to be represented. It also supports definitions of type and subtypes, allowing for efficient extensibility [Meyer 97]. Unfortunately, it may be abused in many ways [Armstrong+94]. In discussing how inheritance can support reuse, Weide notes

> [A] concrete component's implementation must understand the implementation details and subtle representational conventions of all its ancestors in order to implement that component correctly. Unless care is taken, it is possible to introduce components that seem to work properly yet, by manipulating their ancestor's internal data representations, violate subtle and implicit conditions that the ancestors require for correctness [Weide+91, 51].

Inheritance weakens encapsulation, creating a bug hazard similar to global data in procedural languages. It can be used as an unusually powerful macro substitution for programmer convenience, as a model of hierarchy (either problem or implementation), or as a participant in an implicit control mechanism for dynamic binding. It is not unusual to see all three purposes at work in a single class hierarchy. This overloading can lead to undesirable side effects, inconsistencies, and incorrect application behavior. Deep and wide inheritance hierarchies can defy comprehension, leading to bugs and reducing testability.

Inheritance can be used to implement specialization relationships or as a programming convenience. Implementation specialization should correspond to problem domain specialization. Reusability of superclass test cases is predicated on this kind of correspondence. In most cases, convenience subclasses will not reflect a true specialization relationship. As such, it is unlikely they can be excused from testing by testing their superclass, even when a lexical excuse can be found. Fault models and test strategies for inheritance are presented in Section 10.5, Flattened Class Scope Test Design Patterns.

Incorrect Initialization and Forgotten Methods

Inheritance in Objective-C can make it difficult to understand source code [Taenzer+89]. A subclass at the bottom of a deep hierarchy may have only one or two lines of code, but may inherit hundreds of features. Without the aid of a class flattener, the interaction of these inherited features is difficult to understand. In Objective-C, all superclass instance variables are visible in subclasses. This approach creates fault hazards similar to unrestricted access to global data in procedural languages (you can avoid this problem by using accessor/modifier methods instead of statement-level access).

Initialization can easily go awry. Objects are created by the new method. The new method is often inherited and uses a class-specific `initialize` method to actually set subclass instance variable values. Determining how `initialize` is used in a subclass requires examination of the superclass that defines new. The `initialize` message must be sent to `super`, not `self`. Now, suppose new is refined and does not send `initialize` to `self`. Super's `initialize` will not execute, providing incorrect inherited behavior. The compounding effect of polymorphism is discussed later.

When subclasses are composed using many levels of inheritance, proper usage of upper-level features may become obscured. In Objective-C, this approach creates a bug hazard. Cox notes ". . . longstanding bugs have persisted because nobody thought to verify that deeply inherited methods, particularly easily forgotten object-level methods like copy and `isEqual:`, were overridden to adapt to the peculiarities of the subclasses" [Cox 88, 46].

These problems are not limited to Objective-C, but can occur in Smalltalk and Java for the same reasons. Classes that are subclassed from any large, deep framework (e.g., Microsoft's C++ MFC) are also susceptible to these problems.

Testing Axioms

A formal analysis of adequate testing of object-oriented programs is presented in [Perry+90]. Requirements for adequate testing are developed by interpreting Weyuker's software testing axioms (see ***Testing Axioms*** in Chapter 9) for object-oriented programs. Perry and Kaiser's basic result is that intuitive conclusions about test adequacy can be wrong (Berard agrees with these conclusions [Berard 93]).

- Similar functions can require different tests to achieve coverage. For example, a responsibility-based test suite that is adequate for a superclass method may not be adequate when the method is inherited and overridden. This situation arises because a test suite that is adequate for one implementation of a specification is not necessarily adequate for a different implementation of the same specification (antiextensionality axiom).
- Adequate testing of a client does not necessarily result in adequate testing of its servers. For example, suppose a client uses only a few methods of a server object. Even if we try to use the client as a test driver for the server, we cannot exercise all of the server's methods. This situation arises because a test suite that is adequate for a program that calls subroutines is not necessarily adequate for the individual subroutines (antidecomposition axiom).
- Different contexts of usage may require different tests to achieve coverage. For example, a test suite that covers a superclass method may not be adequate for this method in a subclass. This situation arises because test suites that are individually adequate for subroutines are not necessarily collectively adequate for a system that calls these subroutines (anticomposition axiom).

Overriding is typically used to provide superclass/subclass interface consistency while allowing for a different subclass implementation. An overriding method in a subclass can be implemented by a different algorithm, different functionality, or both. The test suite for the overridden method will almost certainly not be adequate for the overriding method.

Changes to a subclass can conflict with unchanged inherited features or unmask superclass faults. Retesting of these unchanged features in the new context is therefore warranted. An exception to this rule occurs with a "pure extension" subclass for which there are "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" [Perry+90, 17].

A detailed analytical procedure to identify the dependencies that determine the scope of retesting is presented in ***Retest Within Firewall.***

Inheritance Structure

A class hierarchy may represent a network of problem domain relationships or share implementation features by some kind of factoring scheme. Attempting to achieve both in a single class hierarchy may lead to bugs [Purchase +91]. Although inheritance can implement type/subtype relationships, this requires care. A "subtype relationship is a relationship between specifications, while a subclass relationship is a relationship between implementation modules" [Leavens 91]. Inheritance may easily be used as code-copying convenience, independent of any relationship between the superclass and subclass specifications.

If an implementation relies on inheritance, its specification-based test suite for inherited features may be reusable, reducing the test effort for a subclass. With the superclass tested, it is necessary to test the subclass against both its own specification and that of its superclass. The subclass is expected to respond to inherited messages in the same way that the superclass does, but with its own data state. Thus the implementation of the superclass may suggest how a subclass might fail. For example, suppose that a subclass is derived from a linked list superclass. The relative values of items for insert and delete operations are unlikely to be affected by faults in the linked list. But suppose the superclass is an ordered tree. In such a case, the relative order matters for correct operations and suggests a testing strategy.

Irregular type hierarchies will lead to incorrect polymorphic message bindings. As Jacobson argues, "We have at least two reasons for an inherited operation not to function in a descendant: (1) If the descendant class modifies instance variables which the inherited operation assumes certain values for, (2) If operations in the ancestor invoke operations in the descendant" [Jacobson+92, 321].

Inheritance will support "accidental reuse," which can lead to bugs [Marick 95, 360]. Suppose class `X` is developed for an application system, but is not designed to be reused—it may have hard-coded assumptions or application-specific performance optimizations. The class has been tested and works without problems in its original application system. Subsequently, however, class `Y` is subclassed from `X` and used in a new application. The superclass methods of `X`, now inherited in `Y`, may produce unanticipated, buggy behavior in the new context, even if the subclass works without trouble.

Multiple Inheritance

With multiple inheritance, a class inherits (directly) from two or more parent classes, which may contain features with the same names. Perry and Kaiser note that multiple inheritance "unfortunately cause[s] very small syntactic changes to have very large semantic consequences" [Perry+90, 18]. Multiple inheritance

presents many bug hazards [Smith+90, Purchase+91, Moreland 94, Chung 94, Ball 95].

- Suppose `Z` is a subclass of classes `X` and `Y`, and method `m` is present in both superclasses. `Z` originally uses `X.m` but is changed to use `Y.m`. `Z` must then be retested. It is likely that the test suite for `Y.m` will not be appropriate for the new `Z.m`.
- An incoming selector may match two or more methods from different superclasses, resulting in incorrect binding and unanticipated interaction among inherited features.
- Repeated inheritance occurs when a common superclass appears in a multiple inheritance hierarchy. For example, suppose classes `B` and `C` are derived from superclass `A`, and class `D` is derived from `B` and `C`. This strategy can lead to multiple symbolic addresses for a single object, aliasing, and inadvertent "self" assignments. If methods and instance variables are not explicitly qualified by class, name clashes can occur and result in unexpected behavior in virtual functions. Pure virtual functions may be renamed and redefined. Public and private inheritance, abstract classes versus concrete classes, and the visibility of superclass data members compound these bug hazards.
- Scoping rules can result in different bindings when superclass methods are used in the context of a subclass. Any code change that involves binding precedence rules requires retesting of superclass methods in the subclass context. A test suite that was adequate for a superclass is therefore not guaranteed to be adequate in a subclass. Similarly, a test suite that was adequate for a baseline configuration may not prove adequate in a new configuration, even if the requirements and interface of the classes have not changed.

Abstract and Generic Classes

An abstract class provides an interface without an implementation [Woolf 97]. Abstract and generic classes are unique to object-oriented programming languages and provide important support for reuse [Meyer 97]. Abstract classes are supported in all six languages considered in this book. Besides the foregoing inheritance bug hazards, they present several specific problems [Thuy 92, Overbeck 94b, Barbey 97].

- We must develop an instantiation to test an abstract class. This process may be complicated if a concrete method uses an abstract method. *Abstract Class Test* presents a fault model.

- A generic class accepts a type parameter(s) that designates a primitive type or class to be substituted for unbound type declarations. The primary bug hazard is unanticipated or incorrect interaction between a type specified as a parameter and the generic class. Firesmith argues that a generic class "may never be considered fully tested" [Firesmith 93b]. Overbeck's testing model demonstrates that generic classes must be instantiated to be tested and to show the limits on reduction of testing for subsequent instantiations [Overbeck 94b]. *Generic Class Test* presents a detailed fault model.

4.2.4 Polymorphism

Polymorphism is the ability to bind a reference to more than one class of objects. In **static polymorphism**, the binding occurs at translation time. For example, Ada 95 types, subtypes, derived types, operators, derived operators, derived subprograms, subprograms from generic instantiations, static types, and tagged types can be bound to two or more objects. C++ templates are another example. The compiler can check the resultant binding and complain when it finds inconsistencies.

Dynamic polymorphism replaces explicit compile-time binding and static type checking by implicit runtime binding and runtime type checking. The semantics, syntax, and mechanisms involved in implementing this type of polymorphism differ for each programming language. Method polymorphism uses **dynamic binding** to select a method to be assigned to a message. The specific method bound to a message is determined when the message is sent. In contrast, static binding assigns a receiver method to a message at compile time. The same polymorphic message can be bound to any sequence of receivers. Each programming language offers its own flavor of dynamic binding and method polymorphism. There are often subtle variations between compilers of the same language.

Polymorphic server classes reduce the size and lexical complexity of client code. The client/server interface is decoupled, facilitating reuse. The method that receives a polymorphic message is determined at runtime by code generated by the language translator. In turn, the argument signature of the message selects the receiver. Sending a polymorphic message is similar to using a case structure to select the receiver of a message, in which each predicate examines the argument signature. Where a case statement includes a hard-coded set of choices, however, the choices for binding are determined by the structure of the

server class at runtime. This mechanism greatly simplifies client interfaces to polymorphic servers and eases maintenance and reuse.

For example, in C++, the **virtual function call** implements method polymorphism. A client sends a message to a server object `fig`, whose class hierarchy is shown in Figure 4.1.

```
void render(Shape &fig ) {
        fig.zbuffer(x,y,z);     // Hide/reveal fig on z axis
        // . . .
}
```

The object `fig` may be a `Circle` or `Triangle`. The virtual function `zbuffer` can be bound to `Shape::zbuffer()`, `Circle::zbuffer()`, or `Triangle::zbuffer()`. The declared base class (`Shape`) is mapped to the object class at runtime. This approach is termed "calling `zbuffer` through the base."

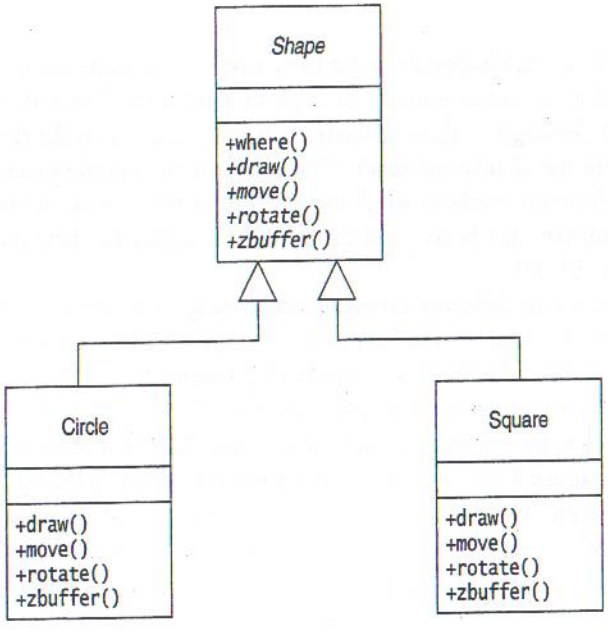


FIGURE 4.1 Shape hierarchy.

Although polymorphism can be used to produce compact, elegant, and extensible code, problematic aspects are identified by many sources [Smith+90, Purchase+91, Thuy 92, Wilde 92, Jacobson+92, Meyer 92a, Wilke 93, Jüttner +94a, Jüttner+94b, McCabe+94, Ponder+94, Moreland 94, Barbey 97]. Subclassing is not necessarily identical to subtyping, so dynamic binding that relies on an irregular class hierarchy can produce undesirable results. Nevertheless, bugs are possible even with well-formed subtypes and formal specification. Suppose superclass method x has been verified. Later, a subclass overrides method x. The correctness of subclass x is not guaranteed because its precondition or postconditions may not be the same as those of superclass x. "Even if the preconditions and postconditions were textually identical, the assertions might have different meanings for each type [class]" [Leavens 91, 78].

Each possible binding of a polymorphic message is a unique computation. The fact that several bindings work without failure does not guarantee that all bindings will work. Object polymorphism with late binding can easily result in messages to the wrong class. It may be difficult to identify and exercise all such bindings.

- Polymorphic, dynamically bound messages can result in hard-to-understand, error-prone code. Such code results in delocalization of plans [Soloway+88] and the yo-yo problem (discussed later).
- Changes can be made to a polymorphic server without regard to clients, assuming that the interface will be correct. If the usage requirements of the server change even a little, an unchanged client may fail.
- Polymorphic message code is deceptively simple. The actual behavior of polymorphic messages is determined by many variables not visible in the client's source code. A developer may find it difficult to imagine all possible interactions under all possible bindings, which create a bug hazard. Even when polymorphic server code is placed side-by-side with a client, imagining just how the server will respond to a particular message can prove challenging. Furthermore, it may be difficult to view the server.
- Unless the server class hierarchy is carefully designed, it can produce strange results for some messages. "What is to prevent a redeclaration from producing an effect that is incompatible with the semantics of the original version—fooling clients in a particularly bad way, especially in the context of dynamic binding? Nothing of course" [Meyer 92a, 47]. The Liskov substitution principle and Meyer's design-by-contract

approach can prevent these problems, but require sophistication and discipline to implement (see Section 10.5, Flattened Class Scope). Server hierarchies designed without regard to these principles are likely to cause client code to fail.

- A message may be bound to the wrong server, if even minor coding errors appear in the client message.

Polymorphic Message Test may be applied to clients; ***Polymorphic Server Test*** may be applied to server hierarchies. ***Percolation*** may be used to implement self-checking polymorphic servers or to serve as a checklist for design and code reviews.

The name overloading and yo-yo problems are not limited to Objective-C programs, but can occur in Smalltalk and Java programs for the same reasons. Classes that are subclassed from any large, deep framework (e.g., Microsoft's C++ MFC) are also susceptible to these problems.

Dynamic Binding

Dynamic binding shifts responsibility for correct operation to the server at runtime. As server classes are often developed and revised independently of clients, the understanding of the server's contract implemented by a client can become inconsistent. A client may assume or require a method not provided by the server, misuse an available method, or construct an interface signature incorrectly.

> It is obvious that the scope for subtle errors with polymorphism is enormous. A debugging agent, unaware of how the + method is overloaded, may have false confidence in a bugged variant after other incarnations pass prescribed tests. Complex polymorphic relations can serve to confuse the implementor, debugging agent, and maintainer alike by, for example, obscuring the point of origin of the bug [Purchase+91, 16].

An experience report on work to rehost a Smalltalk compiler calls this problem a kind of "Catch-22 . . . operations performed are determined [at runtime] from the variable types, and the variable types are deduced [at runtime] from the operations" [Ponder 94]. Methods are typically small (a few lines of code). Many classes may use the same method name. Understandability suffers and a bug hazard results.

> Such systems are also sensitive to subtle naming errors. In a dynamically searched type [class] hierarchy there is a large pool of available procedure names, increasing the chances that a mistyped procedure call will invoke the wrong procedure [Ponder 94, 36].

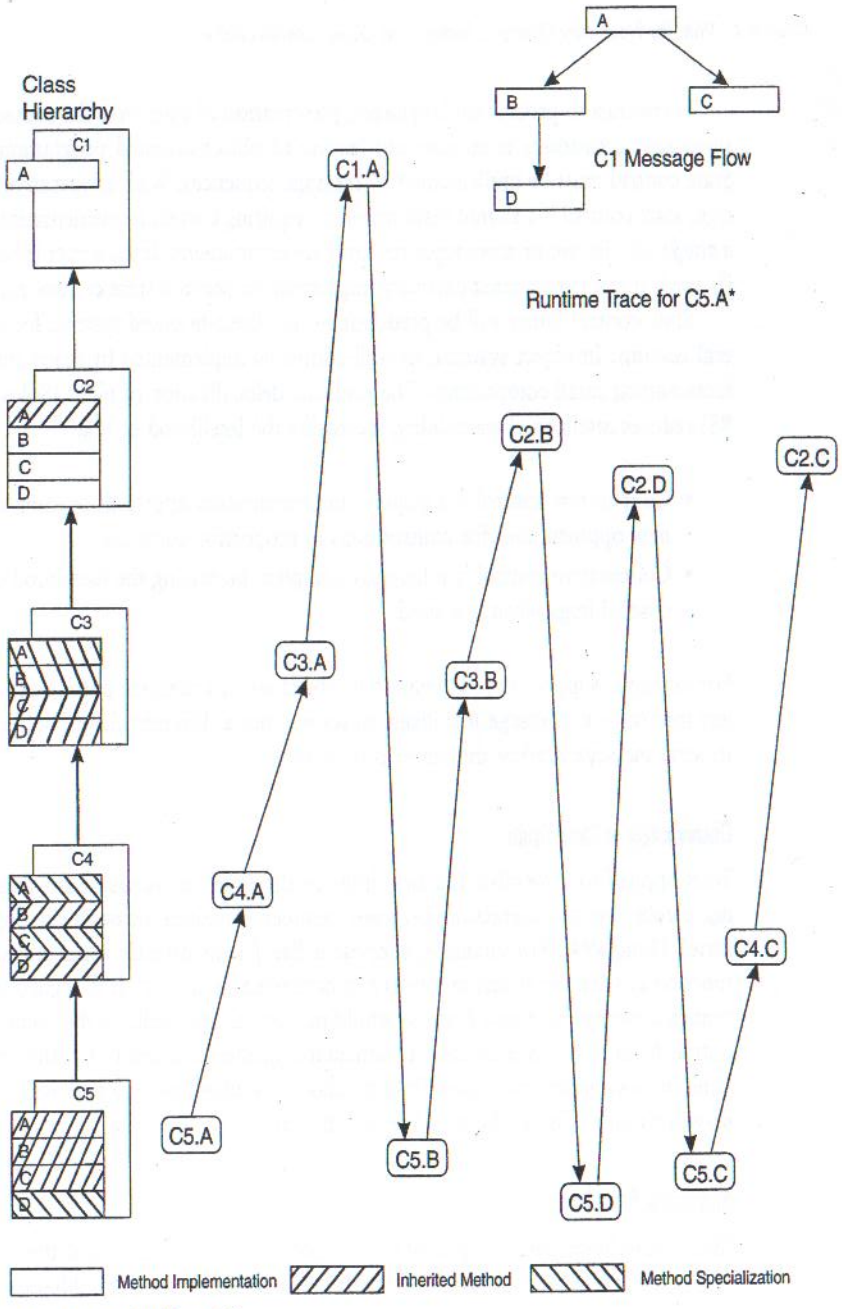
The Yo-Yo Problem

An analysis of reusability suggests many fault hazards due to the message binding mechanisms employed in Objective-C [Taenzer+89]. As classes grow deeper and application systems become wider, the likelihood of misusing a dynamically bound method increases.

> Often we get the feeling of riding a yo-yo when we try to understand one of these message trees. This is because in Objective-C and Smalltalk the object *self* remains the same during the execution of a message. Every time a method sends itself a message, the interpretation of that message is evaluated from the standpoint of the original class. . . . This is like a yo-yo going down to the bottom of its string. If the original class does not implement a method for the message, the hierarchy is searched (going up the superclass chain) looking for a class that does implement the message. This is like the yo-yo going back up. *Super* messages also cause evaluation to go back up the hierarchy [Taenzer+89, 33].

Figure 4.2 illustrates this problem using five classes. `C1` is a superclass, `C2` is its subclass, and so on. In this example, the implementation of method `A` uses `B` and `C`, `B` uses `D`. Messages to these methods are bound according to the class hierarchy and the use of `self` and `super`. "The combination of polymorphism and method refinement (methods which use inherited behavior) make it very difficult to understand the behavior of the lower level classes and how they work" [Taenzer+89, 33].

Table 4.2 shows the difference between lexical message structure and a dynamic trace that can result from it. Suppose an object of class `C5` accepts message `A`. `C5`'s `A` is inherited from `C4`, so the search for `A` begins at `C4.A`. `C4.A` is a refinement, so `C3.A` is checked for an implementation. `C3.A` likewise refers to `C1.A`, where the implementation is found and executed. `C1.A` now sends message `B` to itself, causing `B` to be bound to `self` (an object of class `C5`); the search for `B` therefore begins back at `C5`. An implementation of `B` is found in `C3`. `C3.B` sends `B` to `super` (`C2`). `C2.B` executes, sending `D`, which is again bound to `self` (`C5`). The search for `D` continues up to `C2`, where `D` is implemented. `C2.D` executes for the `C5` object, which then sends message `C` to `self` (still `C5`). The search



*Source: Adapted from [Taenzer 89].

FIGURE 4.2 Trace of yo-yo execution.

TABLE 4.2 The Yo-yo Problem

Class	Method A	Method B	Method C	Method D
C1	Implements Sends self to B and C			
C2		Implements Sends self to D	Implements	Implements Sends self to C
C3	Refines Sends super to A	Refines Sends super to B		
C4	Refines Sends super to A		Refines Sends super to A	
C5				Refines Sends super to A

for `C5.C` is resolved at `C4.C`, which sends it to `super (C3)`. Because `C3` does not implement method `C`, `C2.C` is checked. The implementation `C2.C` is executed on `self` (an object of class `C5`).

Taenzer et al. traced messages in an eight-level class hierarchy and found that a message passed up and down through 58 methods before coming to rest.

4.2.5 Message Sequence and State-related Bugs

The packaging of methods into a class is fundamental to the object-oriented paradigm. As a result messages must be sent in some sequence, raising an issue: which sequences are correct? Which are not? A state-based fault model suggests how state corruption or incorrect behavior can occur. Chapter 7 discusses state models, state faults, and state-based test design in detail. State-based test design patterns are developed in Chapters 10 and 12. State-based testing derives test cases by modeling a class as a state machine. A state model defines allowable message sequences. For example, an `Account` object must have sufficient funds (be in the *open* state) before a `withdrawal` message can be accepted. If the `withdrawal` message causes the balance to become negative, the message causes a transition from the *open* state to the *overdrawn* state. Thus state-based testing focuses on detecting value corruption or sequentially incorrect message responses.

Message Sequence Model

A proposal to generate test cases by an unspecified automatic source code "search" is based on the observation that objects preserve state and typically

accept any sequence of messages [Smith+92]. Four generic bugs that would result in corrupt state are identified:

1. *Interroutine conceptual*: Incorrect design resulting in methods with overlapping responsibilities.
2. *Interroutine actual*: State may be corrupted under certain message sequence patterns.
3. *Intraroutine actual*: State is corrupted due to an incorrect algorithm in a method and incorrect output or abend.
4. *Intraroutine conceptual*: An overriding implementation is omitted or implements an incorrect contract.

Equivalent Sequences

Certain activation sequences of class methods may be expected (specified) to yield identical results. In this situation, the inability of a pair of objects under test to produce equivalent results is deemed a failure. The fault model here is nonspecific: many kinds of faults could cause this kind of failure. The approach is conceptually simple, however, and can be readily automated.

The ASTOOT (A Set of Tools for Object-Oriented Testing) approach is based on "checking whether sequences of operations which, according to the specification, should yield the same state, actually do" [Doong 94].

Member functions that should support associative evaluation are considered fault-prone if "defects have been discovered by applying associativity rules to member functions. That is, if string `s1` is null, and string `s2` is not null, `s1 > s2` should yield the same results as `s2 < s1`" [Fiedler 89].

Smith suggests that an automated "search" of the class under test could yield "identity" sequences [Smith+92]. An identity is an arbitrarily long sequence of messages that should result in no net change to an object. The test compares the initial and final state of the object under test for equality. The mechanics of the search and the test are not described.

Implications of Cooperative Design

Cooperative client/server classes increase the likelihood of state control bugs [Binder 94b]. When used as a server, a cooperative class assumes that the client will not issue incorrect message sequences or send messages with incorrect content [Meyer 92a]. Servers rely on clients to implement some or all of the server's state control. In contrast, a defensive design holds the server responsible for carrying out correct operations and enforcing correct usage of itself under all circumstances. A defensive server is thus tolerant of client faults.

In contrast to procedural languages, preservation of state over the course of successive activations is an essential feature of object-oriented programming. State control must be implemented by message sequences. With cooperative design, state control is a shared responsibility, requiring correct implementation of a single idea in two or more separate software components. If the server is heavily used, many clients must correctly implement the server's state control rules.

State control faults will be predominant in object-oriented systems for several reasons. In object systems, overall control is implemented by many interfaces among small components. The resulting delocalization of plans [Soloway 88] reduces intellectual tractability, increasing the likelihood of bugs.

- Cooperative control is a popular implementation approach opening up new opportunities for control bugs in proportion to its use.
- Cooperative control is inherently complex, increasing the likelihood of control bugs when it is used.

For example, suppose that a client of an object sends a message putting the object into state x. Message p is illegal in state x, but a different client could easily send message p while the server is in state x.

Observability of State Faults

Tests applied to a member function individually "are not adequate for detecting errors due to interaction between member functions through an object state" [Kung+94]. For example, suppose a flag f is incorrectly set in member function a. Member function b uses f to determine an output. It is argued that branch coverage of a and b alone would not reveal this fault, as the incorrect output from b occurs only after a particular sequence of a and b. On the other hand, it is suggested that modeling the class as a state machine and acheiving n-switch cover [Chow 78] would reveal this fault.

Nonspecific State Faults

Many state-based testing approaches are conformance-oriented and therefore rely on a nonspecific fault model. In these approaches, no analysis addresses the specific kind of errors or faults likely to cause the failures that demonstrate nonconformance. Instead, they simply seek to demonstrate that the IUT conforms to its specification. The ASTOOT approach uses the fact that a faulty implementation may not produce the same state after accepting functionally equivalent but operationally different sequences [Doong 94]. Similarly, some sources

adapt (partially) Chow's state-based conformance technique, which is known to reveal certain kinds of control faults [McGregor+93a, McGregor+93b, Hoffman+95]. The OOSE (Object-Oriented Software Engineering) methodology calls for a transition cover of class state models, but does not provide a fault model [Jacobson+92].

4.2.6 Built-in Low-level Services

The general features of object-oriented programming languages are supported by many built-in services. These services are typically automatically generated by the language translator.[2]

- *Default services.* Many compilers will generate default constructors, default destructors, default copy constructors, and default accessors if not explicitly coded. Most developers think that because they didn't code them, they don't need to test them.
- *Conversions.* Problems similar to type conversion with primitive data types (e.g., casting floats to integer) can occur in converting superclass objects to subclass objects, and vice versa. As the type of a referent is not determined until runtime with dynamic binding, changes to a server class can cause unchanged client code to fail. Considerations are discussed in Section 18.3.7, Type/Subtype Equality.
- *Garbage collection.* Built-in garbage collection typically works without problems, but failures can result under high loads.
- *Object identity.* An object-oriented language must provide a unique identity for every object with a runtime scope [Wegner 87]. This identity is rarely a concern of the application programmer, unless object-level operations like copy and equal must be implemented for a subclass. Barbey notes that "identity is not always defined, or observable in object oriented languages, and an oracle may be difficult to build" [Barbey 97, 80]. Relevant considerations are discussed in Section 18.3.7, Type/Subtype Equality.

4.2.7 Bug Lists

The mapping among errors, faults, and failures is complex. As noted earlier, the same general kind of failure can result from many different faults. The same

2. Thanks to Camille Bell for pointing out the problems with default services, conversion, and memory management.

TABLE 4.3 Errors and Failures

Process	Error	Scope A	S	C	M	Failure
Requirements definition	Perceptual: Inadequate problem analysis. Communication failure.	x	x	x		Application problem not solved.
Design	Inadequate problem analysis. Specification error.		x	x	x	Requirements or specification not met.
	Abstraction: Inadequate knowledge of OO design techniques. Premature hierarchic factoring or leveling.		x	x		Poor class structure, bad factoring, inheritance hierarchy sprawling, inconsistent.
	Algorithmic: Specification misunderstood. Implementation incorrect.			x	x	Incorrect method output.
	Inadequate reuse knowledge. Reuse component or hierarchy misunderstood. Incorrect use of polymorphic protocol.	x	x	x		Unexpected, undesired behavior in reused components.
Implementation	Misuse of reuse component, ad hoc extensions of inheritance.			x	x	Incorrect method output.
	Semantic: Inadequate programming knowledge or misuse of target environment services.			x	x	Abend due to type mismatch with late binding.
	Syntactic: Inadequate programming knowledge, typographic error.			x	x	Incorrect output, abend.
Runtime	Inadequate knowledge of application or target services.			x	x	Runtime exceptions, message search failures.

Key: A = application, S = subsystem/cluster, C = class, M = method

general kinds of errors can produce many different faults. Nevertheless, it is instructive to look at these elements separately.

The relationship between errors and failures is represented in Purchase's bug taxonomy [Purchase+91]. This taxonomy was developed to frame a discussion of debugging support for object-oriented code. Bugs are categorized according to process (the activity in which a bug is introduced), failure (the observable incorrect behavior or state), and error (the developer mistake that led to the fault).[3] This classification indicates the circumstances in which bugs are created and how they are manifested. Table 4.3, which is based on this

3. Purchase and Winder's categories were *history* (process), *deviation* (failure), and *mind set* (error).

taxonomy, shows the scope at which these bugs are typically revealed during testing.

Faults lie between errors and failures. While there are an infinite number of particular faults, a short list of general fault types can be compiled. Tables 4.4, 4.5, and 4.6 list fault types at the method, class, and cluster scopes, respectively. These lists are composites that reflect my own experience and three published fault taxonomies: a list of dynamic binding bugs used in a white box testing strategy for a Lisp-based object-oriented language [Trausan-Matu 91]; a list of generic bugs that involve instance variables, methods, messages, classes, inheritance, clusters, and externally driven object interactions [Firesmith 93b]; and design bugs related to abstraction, modularity, and hierarchy that could be detected by a static analyzer [Hayes 94].

These bug lists can be used as checklists for design or code reviews and in the development of test plans. They are not a complete and consistent bug taxonomy, however. Considerable effort would be required to review the

TABLE 4.4 Method Scope Fault Taxonomy

Method Scope		Fault
Requirements		See class scope fault taxonomy (Table 4.5)
		Requirement omission
		Incorrect or missing transformation
Design	Abstraction	Low cohesion
		Responsibilities overlap or conflict with local or superclass method
	Refinement	Feature override missing
		Feature delete missing
	Encapsulation	Overuse of friend/protected mechanisms
		Naked access
	Modularity	Excessively long method
		Method contains no code
		Excessively large instance variables
		Method has low cohesion
	Responsibilities	Incorrect algorithm
		Algorithm inefficient
		Invariant violation
		Exception not thrown
		Exception not caught

Method Scope		Fault
Implementation	General	Message sent to object without corresponding method
		Unreachable code
		Contract violation (precondition, postcondition, invariant)
		Message sent to wrong server object
		Message parameters incorrect or missing, resulting in wrong or failed binding
		Message priority incorrect
		Message not implemented in the server
		Formal and actual message parameters inconsistent
		Syntax error
	Algorithm	Inefficient, too slow, consumes too much memory
		Incorrect output
		Incorrect accuracy, excessive numerical or rounding error
		Persistence incorrect—wrong object saved, or save not done
		Does not terminate
	Exceptions	Exception missing
		Exception incorrect
		Exception not caught
		Incorrect catch
		Exception propagates out of scope
		Exception not raised
		Incorrect state after exception
		Missing object (referred to, but not defined)
	Instance variable define/use	Unused object (defined, but no reference)
		Reference to undefined or deleted object
		Corruption/inconsistent usage by friend function
		Missing initialization, incorrect constructor
		Incorrect type coercion
		Server contract violated
		Incorrect or missing unit (e.g., grams vs. ounces)
		Incorrect visibility/scoping
		Incorrect serialization, resulting in corrupted state
		Insufficient precision/range on scalar type

TABLE 4.5 Class Scope Fault Taxonomy

Class Scope		Fault
Requirements		Requirement omission: missing use case, and so on
		Problem domain object not identified
		Incorrect scope of effort, system boundaries
		Incorrect or inappropriate level of abstraction used
		Incorrect state model
		Association missing or has incorrect multiplicity
Design	Abstraction	Redundant allocation of responsibilities; the same method appears in several subclasses without being defined in a common superclass
		Design of top-level objects and protocol; poor class structure, nonorthogonal protocols; inheritance hierarchy sprawling, inconsistent
		Association missing or incorrect
		Incorrect type parameter(s) specified for a generic class
		Inheritance loops—for example, with classes A, B, and C, A hasSuper C, B hasSuper A, C hasSuper B
	Refinement	A subclass incorrectly redefines a superclass method
		Wrong feature inherited
		Subclass method to override missing
		Subclass method to nullify superclass method missing
		Hierarchy does not conform to LSP
		Incomplete specialization, missing methods or variables
		Deferred resource not provided by subclass of deferred superclass
		Dead-end or cycle in hierarchy
		Incorrect multiple inheritance
		Improper placement of class
	Encapsulation	Public interface to class not via class methods
		Implicit class-to-class communication
	Modularity	Overuse of friend/protected mechanisms
		Private/protected method not used
		Object not used
		Excessively large number of methods
		Too many instance variables
		Module, class, method, instance variable not used
		Module contains no classes
		Method contains no code

Class Scope		Fault
		Class contains no methods
		Class contains no instance variables
		Excessively large instance variable
		Excessively large module
	Responsibilities	Incorrect state model
		Contract inconsistent
		Incorrect specification of concurrency
		Invariant violation possible
		Association missing or incorrect
Implementation		Incorrect method under multiple inheritance, due to synonyms, naming error, or misuse of name resolution
		Incorrect constructor or destructor
		Incorrect parameter(s) used in generic class
		Abstract class instantiated
		Syntax errors
		Association not implemented
Process		Class documentation inconsistent with its implementation
		Class doesn't meet applicable design or code standards
		Incorrect version/build used

categories and produce a taxonomy that complies with the *IEEE Standard Classification for Software Anomalies*, for example.

4.3 Language-specific Hazards

Most of the foregoing general bug hazards are present in the six languages discussed in this book, with obvious exceptions. For example, multiple inheritance bugs cannot occur in Java, Objective-C, or Smalltalk because these languages do not support multiple inheritance. Specific problems noted by users of these languages provide additional insight. The absence of sections on Ada 95, Eiffel, and Objective-C does not imply anything in particular about these languages, other than that I am not aware of published reports that detail hazards specific to these languages.

TABLE 4.6 Cluster/Subsystem Fault Taxonomy

Cluster/Subsystem Scope		Fault
Requirements		Requirement omission
		Incorrect abstraction
		Incorrect state model
		Association missing
Design	Encapsulation	Implementation data structure incorrectly exported
		Incorrect export of feature
		Incorrect package visibility
		Class contains nonlocal method
		Public method not used by object users
		Message/object mismatch
		Incorrect environment interface
	Modularity	Missing component
		Inconsistent component
		Low cohesion among components
Implementation		Incorrect priority
		Incorrect serialization
		Message sent to destroyed object
		Inconsistent garbage collection
		Incorrect message/right object
		Correct exception/wrong object
		Wrong exception/right object
		Incorrect resource allocation/deallocation
		Concurrency problems
		Inadequate performance
		Deadlock

4.3.1 C++

Stanley Lippmann, who with Bjarne Stroustrup was a lead developer of the C++ language, reflected on its difficulties in a recent interview.

> DR DOBBS JOURNAL: You worked at AT&T Bell Labs, in a research environment. You are now at Disney. Have you learned anything about the language now that you are yourself a user, working in "the real world" with other users of the language?
>
> STANLEY LIPPMANN: Actually it was a major insight to me. I think of it as a sort of Fairfield symmetry, if you will. I worked with Bjarne, Andy Koenig and other really smart people. We all understood the language reasonably well and when it was complicated, we understood why it was complicated—to accommodate C, for example. When I moved to Disney, I found myself working with PhDs in graphics and mathematics who were, again, wonderfully smart, but they didn't understand C++. They would actually program, if you will, "naively." I was suddenly stunned by how difficult C++ can be and by how it can become an obstacle to people getting their work done [Dr Dobbs 98].

Sakkinen notes that C++ "inherits" some deficiencies of C. Understanding the behavior of arrays, constructor/destructors, and virtual functions requires consideration of complex special cases and counterintuitive behavior [Sakkinen 88]. "C++ is a powerful language that facilitates writing clear, concise, efficient, and reliable programs. However, its many features can be bewildering to the uninitiated, and injudiciously used, they can lead to programs that are confusing, bloated, slow, and bug-ridden" [Shopiro 93, 211]. Reed observes that "Most C++ bugs are due to unsafe features of the language: pointer problems, memory problems, uninitialized values, improper casts, improper union usage" [Reed 93].

The memory leak problem is not a problem exclusive to C++. Nevertheless, some of my clients estimate that half of their C++ failures are caused by memory leaks. The use of assertions in base classes, constructors, and destructors is recommended to reveal memory allocation and pointer corruption faults [Thielen 92, Spuler 94, Cline+95, Stout 97, Payne+97]. (See Chapter 17, Assertions.)

Fiedler notes that the object under test may be used as an input parameter to itself. Suppose `s1` is a string. Then `s1.append(s1)` may reveal pointer faults or insufficient isolation of self-referents [Fiedler 89]. Incorrectly duplicated instance variable names (pointer aliases) are easily made errors that can prove difficult to detect. Aliasing occurs when "an object, or part of it becomes globally accessible. As a result, it is possible to change the state of an object without calling any of its routines" [D'Souza 94, 34].

A list of 50 C++ "gotchas" provides a detailed catalog of easily made errors [Cargill 93]. These bugs typically result when the developer fails to consider the implication of some subtle C++ semantics. For example, in the following code fragment, only `Base::~Base` is called upon the deletion of `Derived`. Because the destructor for `Derived` is not activated, its allocated memory becomes garbage (this situation can be prevented by declaring the base destructor as virtual). Simple class extensions can lead to hard-to-find bugs. Similar problems occur when a member function is added to a base class that overrides a function of the same name defined at the file scope. Derived class objects will not be bound to the global function, possibly resulting in a failure [Reed 94].

```
// Wrong destructor called, scoping bug
class Base {
public:
   ~Base();
...
}

class Derived : public Base {
public:
   ~Derived();
...
}

void client::code()
{
   Base *p = new Derived;
   ...
   delete p;
   ...
}
```

Firesmith, Davis, and Moreland report similar kinds of common C++ bugs:

- (1) Misuse of friend functions, misuse of private, protected, and public scope, and incorrect type casts resulting in conflicting access or unexpected binding; (2) References in a base class to a derived class; (3) Pointers, arrays, unions, and casts (type conversions) that defeat type-safe usage; (4) Omission of virtual declarations in polymorphic superclasses; (5) Ambiguities in overloading public and private members; (6) Assignment taking place other than at initialization; (7) Overloading that ignores returned types; (8) Incorrect object references (size and address) [Firesmith 93b].
- (1) Absence of input parameter range checking; (2) Using an incorrect `this` pointer; (3) Using an incorrect object pointer for a virtual class; (4) State corruption due to bugs in constructor, destructor, or invalid array references [Davis 94].
- (1) Local names that hide global names; (2) Overloading and defaulting that are similar in appearance but have very different semantics; (3) Implicit type coercion with overloaded operators [Moreland 94].

4.3.2 Java

A list of 36 Java "gotchas" details inconsistent and surprising results [Green 98]. For example,

- A superclass method or variable name can be overridden in a subclass or shadowed. A shadowed superclass instance variable is declared in the subclass with the same name, but subclass references prefix the superclass name to the instance variable name. The resultant binding is resolved in one of four ways: static method, instance method, static variable, or instance variable.
- Some syntax similar to C++ is used, but with subtle differences. Developers who are accustomed to the C++ semantics are likely to misuse these constructs.
- The + operator invokes either addition (scalars) or concatenation (strings). Java determines the operation based on the operand type, "but humans can be easily fooled."

Java applets are susceptible to portability bugs and platform-specific differences in each Java Virtual Machine (JVM). A portability bug causes the applet to fail (or run) on some JVMs but not on others. This bug may appear in the applet, a particular JVM, or both. Applet bugs result from nonstandard or incorrect Java code. Enough variation exists among JVMs to warrant testing applets on different JVMs [Hayes 97]. Some of these differences result from the Java language specification, which does not define a specific thread scheduling policy, legal size of file names, or number of pixels on the screen. Different physical clients using the same JVM may see different behavior (bugs) due to varying link speeds and differences in firewalls and security configuration.

4.3.3 Smalltalk

Johnson has compiled a list of "classic" Smalltalk bugs [Johnson 97]. Table 4.7 provides an abbreviated version of this list. Interested readers are encouraged to locate the original version on the Web. This list suggests how some of the essential features of the Smalltalk object model have the unintended consequence of being bug hazards.

Silverstein notes that Smalltalk applications have several intrinsic bug hazards [Silverstein 99]. In addition to the problems noted earlier regarding polymorphism and violations of the superclass contract, "weak typing permits sloppy use of interfaces." Initialization bugs arise when a subclass neglects to override a superclass method (#copy), omits initialization code, or uses #new (lazy initialization). Despite built-in garbage collection, unreleased instances can lead to memory leaks. Bugs can be introduced during application

TABLE 4.7 Johnson's Classic Smalltalk Bugs

	Bug	Cause
1	Modifying elements of variable-sized collection classes	Superclass copy methods do not support subclass extensions. When new instance variables are added to a collection subclass, corresponding code to copy these variables must be written.
2	Using `add:` as if it returns a collection	Mutable collections use `add:`, which returns its argument, not the receiver collection.
3	Changing collection while iterating over it	Modifying a collection while iterating through its elements will cause elements of the collection to be moved. Elements might be missed or handled twice.
4	Copy of a collection is modified, but original is not	A modifiable copy of a collection is instantiated, but the modifications are not also made to the original.
5	Missing ^	Incorrect object returned.
6	Incorrect instance creation method	Incorrect object returned, incorrect usage of new, `super new init`, and so on.
7	Assigning to classes	A class will accept an assignment, causing the class to be corrupted.
8	Use of `become:`	After `become:` executes, the specified variable names are the same, but point to different objects. This process can wreak algorithmic havoc. Semantics of this operation differ among Smalltalk implementations.
9	Recompiling in Smalltalk/V	Smalltalk/V will accept references to undefined objects without complaint.
10	Opening windows	Smalltalk/V and the older versions of Smalltalk-80 do not return a window object to the sender when a new window is opened. Clients' attempts to use the object returned by the open window will fail.
11	Incorrect use of blocks	Block scoping (binding) of temporary variables and precedence rules can defeat the programmer's intent.
12	Cached menus	Menus are often defined in a class method, where they are created. The creation message does not necessarily result in initialization, however.
13	Incorrect cascaded messages	The omission of parentheses in a cascaded message statement causes the first message's result to be overwritten (instead of returned) to the next message.
14	Using class methods to implement a singleton object	Class methods used to implement global variables are easy to misuse and often cause maintenance problems.
15	Redefining = but not redefining `hash`	A redefinition of `hash` is required to produce the correct object identity.

packaging as a result of name overloading—for example, with the `#add:` and `#do:` methods.

> Collaboration and distribution of behavior increase dependencies between objects. Coupling between objects implies that a fix for one problem may cause a new problem—sometimes in an unanticipated area. This most often happens when fixes are applied late in a development cycle when there is little time to think about the broader implications of changes or perform extensive regression testing [Silverstein 99].

Besides recommending early class testing, integration testing, and regression testing, [Skubics+96] advocates roughly 100 specific design and coding guidelines. Many of these guidelines imply a bug hazard to be avoided. For example, guideline 98, "Do not override the identity == or ~~ operations," is given because such an override can "change the fundamental behavior of the Smalltalk system" [Skubics+96, 79].

4.4 Coverage Models for Object-oriented Testing

Coverage is an operational definition for a complete test suite. A fault model suggests a coverage goal, as we want to exercise all elements that are likely to be buggy. For example, if a fault exists, it must be exercised to be revealed, so achieving statement coverage satisfies this simple fault model. Statement coverage is defined as the percentage of all executable source code statements in a program that have been exercised at least once by a test suite. The details of code coverage are discussed in Section 10.2.2, Implementation-based Test Models.

Many researchers and practitioners call for "thorough" testing without reference to any explicit coverage criteria. In the absence of a commonly understood operational definition of "thorough" testing (e.g., 100 percent branch coverage), the only meaningful interpretation of a "thorough" test suite is that it has satisfied someone's curiosity.

4.4.1 Code Coverage and Fault Models

The scope of retesting under inheritance established in [Perry+90] is widely, if sometimes reluctantly, accepted. It defines a general scope of testing, but does not advocate a specific coverage based on a probable fault model (see Section 4.2.3). Nevertheless, this analysis has had a significant influence on subsequent work; it is discussed in [Smith+92, Harrold+92, Murphy+92, Jacobson+92,

Klimas 92, Berard 93, Coleman+94, Graham 94, D'Souza+94, Turner+95, Barbey 97]. As Perry's result requires complete testing, at least one technique has been developed to reduce the number of method-specific tests required for derived classes [Harrold+92]. A survey of abstract data type coverage definitions appears in [Binder 96a]. A coverage model for intraclass control and data flow is presented in The Class Flow Graph: A Class Scope Coverage Model section, in Chapter 10 (pages 389–396).

Optimistic Scope

An optimistic view of inheritance argues that it will reduce the testing necessary in derived classes, provided that the base classes are "thoroughly" tested [Cheatham+90]. Testing of base classes is similar to unit testing of modules in procedural languages. Unit testing may begin "as soon as a class is implemented. If the class is not a derived class, the unit testing is equivalent to unit testing in a traditional system. . . . If the class being tested is a derived class, the parent class should be thoroughly tested first. Then the derived class can be tested in conjunction with the base class" [Cheatham+90].

The assertions made in Cheatham's paper reflect an optimistic appraisal of the benefits of object-oriented development. Many of them are contradicted by later results. Message passing and inheritance influence how much testing will be needed. With unaltered inheritance, "little additional testing [of a derived class] is needed. At most the interface should be retested." An overridden or unique method "must be tested as a new member function." The meaning of "thoroughly tested" is not discussed and the assertions about testing scope are not supported.

It is speculated that reuse of "thoroughly" tested classes may reduce unit testing by 40 to 70 percent. System testing should also be eased because requirements are more readily mapped into implementation (compared to procedural implementations), but "further study is needed."

Method Scope Branch Coverage

Several variants of branch coverage [Myers 79] and basis-path coverage [McCabe 76] have been used or advocated. "Test cases are created to execute each decision path in the code. . . . except for paths that contained code for exception handling, test cases were written to ensure complete path coverage of each member function" [Fiedler 89].[4]

4. It is likely that by "complete path coverage" Fiedler means 100 percent branch coverage, and not all entry–exit paths.

Jacobson calls for testing each "decision-to-decision path" (a path between two predicates) at least once, noting that exercising pairs of decision paths may be useful (but time-consuming) [Jacobson+92]. Berard adds interrupts and exceptions: "Enough test cases must be written so that we can be reasonably assured that all statements are executed at least once, all binary decisions take on a true and false outcome at least once, all exceptions are raised at least once, and all possible interrupts are forced to occur at least once" [Berard 93, 259].

Coverage for a Classless Language

Two adequacy criteria are given for testing programs written in mXRL, a classless LISP-based object-oriented language [Trausan-Matu+91]: component (similar to statement coverage) and message (similar to branch coverage). "In addition to the testing of procedural components, in OOP we have to define also procedures for testing the correctness of inheritance paths." Adequate testing of a class requires the following: (1) all statements in each method must be exercised, (2) all locally defined (not inherited) attributes "must be accessed," and (3) each feature obtained by a one-step inheritance link must be used at least once. It is asserted (without substantiation) that, compared with procedural testing, "Due to code sharing by inheritance, the amount of tests corresponding to the statement adequacy criterion may be significantly reduced."

Branch coverage for mXRL is similar to covering "all the different possibilities of code activation (i.e., branches)." Message coverage requires (1) exercising all branches in methods, (2) sending all "the possible different types of messages" that an object can accept, (3) sending a given message to all objects that accept the particular message, (4) exercising all inheritance links, including implicit (multilevel) inheritance links, and (5) exercising all possible exits from methods composed by multiple inheritance.

ICpak Test Suite

The coverage goals set and met for testing of the Objective-C ICpak class library considered the effects of inheritance and dynamic binding. Each subclass was tested to demonstrate that "every method in all its superclasses was executable and gave correct results."

> With an object-oriented language one must verify that: all inherited methods used by a class are correct, all arguments that are subclasses of the specified argument type are correct, all methods by the same name perform the same logical operation, [and] the documentation is accurate and sufficient for an isolated user to use all the components [Love 92, 193].

Coverage Checklist

Classes are not testable per se, but class testing is crucial for reuse. Consequently, "testing should be thorough . . . completely unit testing a single instance completely unit tests its corresponding class" [Firesmith 93b]. Concrete, abstract, deferred, and generic classes are all subject to unique bugs. Generic classes (e.g., C++ templates) "may never be considered fully tested."

> Changes to superclasses may have unexpected effects on the class to be tested. Changes to a superclass may therefore obsolete test results and require significant regression testing, especially if configuration management and detailed analysis do not rule out impacts on the class to be tested. . . . The primary purpose of classes and inheritance is reuse. For this reason, initial testing should be exhaustive [sic] and include stress testing [Firesmith 93b].

Whereas the "testing of objects is application-specific, the testing of classes should be more general because the developer of a class cannot know in advance how the instances of that class may be used on future projects." Firesmith argues that a complete class test suite requires the following:

- Every operation is executed.
- All message parameters and exported attributes are checked using equivalence class samples and boundary values.
- Every outgoing exception is raised and every incoming exception is handled.
- Every variable attribute is updated.
- Every state is achieved.
- Every operation is executed in each state (correctly where appropriate, prohibited where inappropriate).
- Every state transition is exercised to test assertions.
- Appropriate stress, performance, and suspicion tests are performed.

Incremental Class Testing

Harrold et al. suggest that a minimal set of inherited C++ derived class features to be tested can be determined automatically [Harrold+92]. Testing of base and derived classes is therefore monitored. The selection criteria is argued to be consistent with [Perry+90], but aims to reduce or reuse base class tests. A "thorough" test suite is manually prepared and applied to each base class, then each

derived class is flattened. Required test cases for derived classes are determined by

> incrementally updating the [testing] history of the parent class to reflect the differences from the parent. Only new attributes or those inherited, affected attributes and their interactions are tested. The benefit of this technique is that it provides a savings both in the time to analyze the class to determine what must be tested and in the time to execute test cases [Harrold+92, 78–79].

A new feature is "thoroughly" tested at the first level at which it appears. If a derived class has a new feature, it is "thoroughly" tested in the context of its defining class. "We first test base classes using traditional unit testing techniques to test individual member functions in the class." Server methods are stubbed out. The specific test case design technique is not discussed, nor is an operational definition of a "thoroughly" tested class provided.

Decision rules for setting the scope of derived class tests are presented. For a new or untested feature in a derived CUT, a complete test suite is developed. Intraclass and interclass integration tests are prepared. A new data member is tested by "testing [the CUT] with member functions with which it interacts."

For inherited features, "very limited retesting" is needed, since it is argued that the specification and implementation are assumed to be identical. Integration testing is indicated by the anticomposition axiom if the CUT interacts "with new or redefined variables, or accesses the same instances in the class's [sic] representation as other member functions."

For redefined features, reusing the test cases prepared from specifications may be possible, but new implementation-based test cases become necessary. Conditions under which method-specific tests need not be repeated for inherited methods are presented.

It is argued that this approach can significantly reduce the number of tests needed to verify inherited features while still meeting Perry's adequacy axioms. This approach is referred to as Hierarchic Incremental Testing (HIT) in several subsequent reports.

Class Interface Data Flow Coverage

Zweben models a class interface with a graph whose nodes represent "define" and "use" methods. The define/use ADT operations developed through this data flow test strategy may be applied to classes [Zweben+92]. (See Chapter 10 for more on the data flow model.) With the flow graph, test sets can be identi-

TABLE 4.8 Method Define/Use Coverage

Code Coverage Criterion	Corresponding Class Interface Coverage
All statements	All operations
All branches	All feasible edges
All paths	All paths
All uses	All uses
All definitions	All definitions
All DU paths	All DU message sequences

fied that provide covers corresponding to various data flow coverage criteria (Table 4.8).

A similar approach synthesizes a data flow model from a class interface and advocates coverage based on several variations of set (define) and get (use) sequences [Parrish 93]. As the flow graph may contain infeasible edges due to conditional flow, coverage may be unobtainable. A "weak class graph" is defined where conditional edges are treated as a single edge (a "weak" edge), assuring that at least one of the conditional edges can be covered.

Polymorphic Bindings

Dynamic binding presents a coverage problem. Based on experience developing a 3 MLOC C++ system, Thuy argues that exercising a single binding of a polymorphic server is "insufficient: the coverage is complete only when all the redefinitions of the called method have also been exercised" [Thuy 92]. This approach, however, could require a large number of test cases, which may be difficult to identify.

McCabe proposes a component-directed strategy to deal with the large number of tests needed to cover polymorphic servers [McCabe+94]. Each class in the system under test is tested separately by a driver. The class test suite must (1) provide basis-path coverage, (2) exercise all intraclass uses of class methods, and (3) exercise each overloading of a polymorphic server at least once. When a class passes these tests, it is considered "safe." It is argued that tests for a client of safe server classes need not retest polymorphic bindings encapsulated in the servers. It is also asserted that regression testing for clients of safe classes is unnecessary in some circumstances. If the implementation of a safe class changes (but not the interface or behavior), no retesting of the client is needed. No empirical or analytical support is provided for these assertions.

4.5 An OO Testing Manifesto

Effective object-oriented testing must address both technical and process issues. Several observations can be made:

- The hoped-for reduction in object-oriented testing due to reuse is illusory.
- Inheritance, polymorphism, late binding, and encapsulation present some new problems for test case design, testability, and coverage analysis.
- To the extent that object-oriented development is iterative and incremental, test planning, design, and execution must be similarly iterative and incremental.
- Regression testing and its antecedents must be considered a sine qua non for professional object-oriented development.

So, while nearly all of what we know about testing procedural and abstract type software applies, testing object-oriented software has significant differences. Effective testing of object-oriented software must be guided by three basic tenets (supporting propositions for each tenet follow the list):

1. *Unique bug hazards.* Test design must be based on the bug hazards that are unique to the object-oriented programming paradigm. These techniques must be augmented with applicable established test design practices.
2. *Object-oriented test automation.* Application-specific test tools must be object-oriented and must offset obstacles to testability intrinsic to the object-oriented programming paradigm.
3. *Test-effective process.* The testing process must adapt to iterative and incremental development and mosaic modularity. The intrinsic structure of the object-oriented paradigm requires that test design consider method, class, and cluster scope simultaneously.

Unique Bug Hazards

The object-oriented programming paradigm is significantly different from the procedural, functional, or abstract type paradigms. It is error-prone in unique

ways. Effective test design of object-oriented software must rely on fault models that recognize these hazards.

1. The interaction of individually correct superclass and subclass methods can be buggy. These interactions must be systematically exercised.

2. Omitting a subclass override for a high-level superclass method in a deep inheritance hierarchy is easy. Superclass test suites must be rerun on subclasses and constructed so that they can be reused to test any subclass.

3. Unanticipated bindings that result from scoping nuances in multiple inheritance and repeated inheritance can produce bugs that are triggered by only certain superclass/subclass interactions. Subclasses must be tested at flattened scope, and superclass test cases must be reusable.

4. Poor design of class hierarchies supporting dynamic binding (i.e., polymorphic servers) can result in failures of a subclass to observe superclass contracts. All bindings must be systematically exercised to reveal these bugs.

5. The loss of intellectual control that results from spaghetti polymorphism (the yo-yo problem) is a bug hazard. A client of a polymorphic server can be considered to have been adequately tested only if all server bindings that the client can generate have been exercised.

6. Classes with sequential constraints on method activation and their clients can have control bugs. The required control behavior can be systematically tested using a state machine model. Testing that is not designed to exercise control systematically cannot be expected to reveal these bugs.

7. Subclasses can accept illegal superclass method sequences or generate corrupt states by failing to observe the state model for superclass. Where sequential constraints exist, subclass testing must be based on a flattened state model.

8. A generic class instantiated with a type parameter for which the generic class has not been tested is almost the same as completely untested code. Each generic instantiation must be tested to verify it for that parameter.

9. The difficulty and complexity of implementing multiplicity constraints can easily lead to incorrect state/output when an element of the composition group is added, updated, or deleted. The implementation of multiplicity must be systematically exercised.

10. Bugs can easily hide when a set method computes a corrupt state, the class interface does not make this corrupt state visible, and the corruption does not inhibit other operations. Define/use method sequences must be systematically exercised to reveal this kind of bug.

11. Mosaic modularity and the resulting delocalization of plans are intrinsic to the object-oriented programming paradigm. Control relationships at a scope beyond the class are obscured. OOA/D control abstractions at this scope are weak, offering only fragmentary models. This bug hazard calls for systematic testing based on integrated control models at this scope.

Object-oriented Test Automation

The structure of a test automation system must be shaped by the abstractions of the system under test and by its implementation dependencies.

1. The encapsulation and mosaic modularity intrinsic to object-oriented languages decrease controllability and observability. Any given implementation under test consists of many small objects, which contain many small objects, which contain many small objects, and so on. Their class interfaces typically do not support all of the state set/get requirements of systematic testing. Test automation countermeasures are required in proportion to encapsulation and scope of testing.

2. The absence of design for testability in large systems can greatly reduce testing effectiveness. Not only do controllability and observability suffer, but code dependencies can also force testing to be conducted at a larger (and less effective) scope than might otherwise have been prudent. Design for testability must be considered at all stages of development.

3. Test drivers and test suites should correspond to the inheritance, cluster, and package structure of the system under test. This tactic improves test suite maintainability and test suite reuse.

4. Design-by-contract implemented with assertions is a straightforward and effective approach to built-in test. Not only does this strategy make testing more efficient, but it is also a powerful bug prevention technique.

5. As of publication, no COTS tools support coverage analysis at any scope beyond the method and static call-pair. This is woefully inadequate for testing object-oriented code. I hope object-specific coverage analyzers will become available in a few years. Meanwhile, existing coverage tools must be used.

Test Process

The work of testing must be orchestrated to complement the technical constraints and opportunities of the object-oriented paradigm and the structure of the system under test.

1. Although a class is the smallest natural unit for testing, class clusters are the practical unit for testing. Testing a class in total isolation is typically impractical. Methods are meaningless apart from their class, but must be used individually to exercise class responsibilities. Method testing must consider the class as a whole. As a result, the test design must be method-specific but based on cluster-scope responsibilities. Tests must be applied in a sequence determined by implementation dependencies. At this scope, testing must combine aspects of unit and integration testing.

2. Test design techniques must use available OOA/D models at class, cluster, and use case level. Test models developed to compensate for missing, incomplete, or incorrect requirements and design models should follow known-good practices for OOA/D. This approach allows the same test design strategies to be used in either circumstance.

3. Human checking of design and code work-products may be less effective (compared with procedural code). In procedural systems, static verification (walkthroughs or inspections) has been shown to be more effective and efficient in removing certain kinds of faults than testing [Basili+87]. However, object-oriented source code can be harder to comprehend for at least three reasons: (1) the yo-yo problem, (2) dynamic binding, and (3) cooperative distributed control strategies. Given these obstacles to comprehension, static techniques may miss bugs, requiring more testing to achieve comparable quality.

4. The test strategy for producers of reusable components is different from the reuse consumer's test strategy as well as from testing of software that is not intended for general reuse. These strategies are discussed in Chapter 11.

5. Although unfounded expectations about the magical charms of object-oriented programming have diminished in recent years, hope springs eternal. Some developers ignore or disparage known-good software engineering practices, including testing. This attitude must be confronted and corrected.

6. Class testing must be closely tied to class programming. Object-oriented unit testing proceeds in a shorter cycle than the corresponding activities in procedural development. The development process must facilitate short code/test

cycles. Not all test activities will follow this pattern, however. Subsystem integration and system test cannot be performed until some or all components are in hand. These test activities will continue to be final steps in object-oriented development.

Just as some aspects of object-oriented design and programming have proved to be difficult, so are some aspects of testing object-oriented systems. However, if we are to master the paradigm and realize its promise, these challenges must be met.

4.6 Bibliographic Notes

This chapter includes material adapted from [Binder 94a, Binder 95b, Binder 96i, Binder 97c].

For an in-depth comparative analysis of programming language paradigms, see [Watt 90]. An extensive list of research reports on bug rates, complexity measurement, and effort estimation can be found at *Object-Oriented Metrics: An Annotated Bibliography,* maintained by the Empirical Software Engineering Research Group, Department of Computing, Bournemouth University, United Kingdom, at

http://dec.bournemouth.ac.uk/ESERG/bibliography.html