Chapter 3 Testing: A Brief Introduction

> How strange it is to say that testing a program and never having it result in a failure is a problem, but indeed that is exactly what we are saying.
>
> *Friedman and Voas*

Overview

The chapter presents a point of view on software testing. Basic terms are defined and linked to chapters and sections where they are discussed. The general limits and uses of testing are discussed.

3.1 What Is Software Testing?

I view software testing as a problem in systems engineering. It is the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs. Tests are designed by analyzing the system under test and deciding how it is likely to be buggy. In turn, test design provides the requirements for the test automation system. This system automatically applies and evaluates the tests. It must be designed to work with the physical interfaces, structure, and the runtime environment of the system under test. Manual testing, of course, still plays a role. But testing is mainly about the development of an automated system to implement an application-specific test design.

Effective testing cannot be achieved without using abstraction to conquer the astronomical complexity of typical software systems. Test design must be

based on both general and specific models. General models offer a systematic and repeatable means to generate test suites. Combinational logic (Chapter 6) and state machines (Chapter 7) provide the general models used in this book. You can view these models as reusable functions, to which application-specific relationships are input. In addition, application-specific test models must be developed to represent the required behavior of the system under test. A testable OOA/D model of the application can be used, if one is available. If not, application-specific test models must be developed. Chapter 8 discusses application-specific modeling. The test generation algorithm of the general model produces the application test suite as an output. This output serves as the input to the test automation system, which in turn evaluates a test run as pass or no pass. The design of test automation presents its own challenges, which are sometimes exotic. Figure 3.1 shows these relationships.

Test design requires solving problems similar to those encountered in the analysis, design, and programming of an application system. Test models are developed to represent responsibilities. Because tests must be executable, however, the equivocal abstraction that greases the wheels of analysis and design is not acceptable.

Test design involves several steps:

1. Identify, model, and analyze the responsibilities of the system under test.
2. Design test cases based on this external perspective.
3. Add test cases based on code analysis, suspicions, and heuristics.
4. Develop expected results for each test case or choose an approach to evaluate the pass/no pass status of each test case.

General solutions to recurring test design problems are presented as test design patterns in this book.

After design is complete, the tests are applied to the system under test. Unless manual testing is indicated, a test automation system must be developed to run the tests. This system may be either simple or complex. It may be implemented with general-purpose test tools (scripting, coverage analyzers, and so on), by coding application-specific test drivers and stubs, and by adding test code to the application. A test automation system typically will start the implementation under test, set up its environment, bring it to the required pretest state, apply the test inputs, and evaluate the resulting output and state.

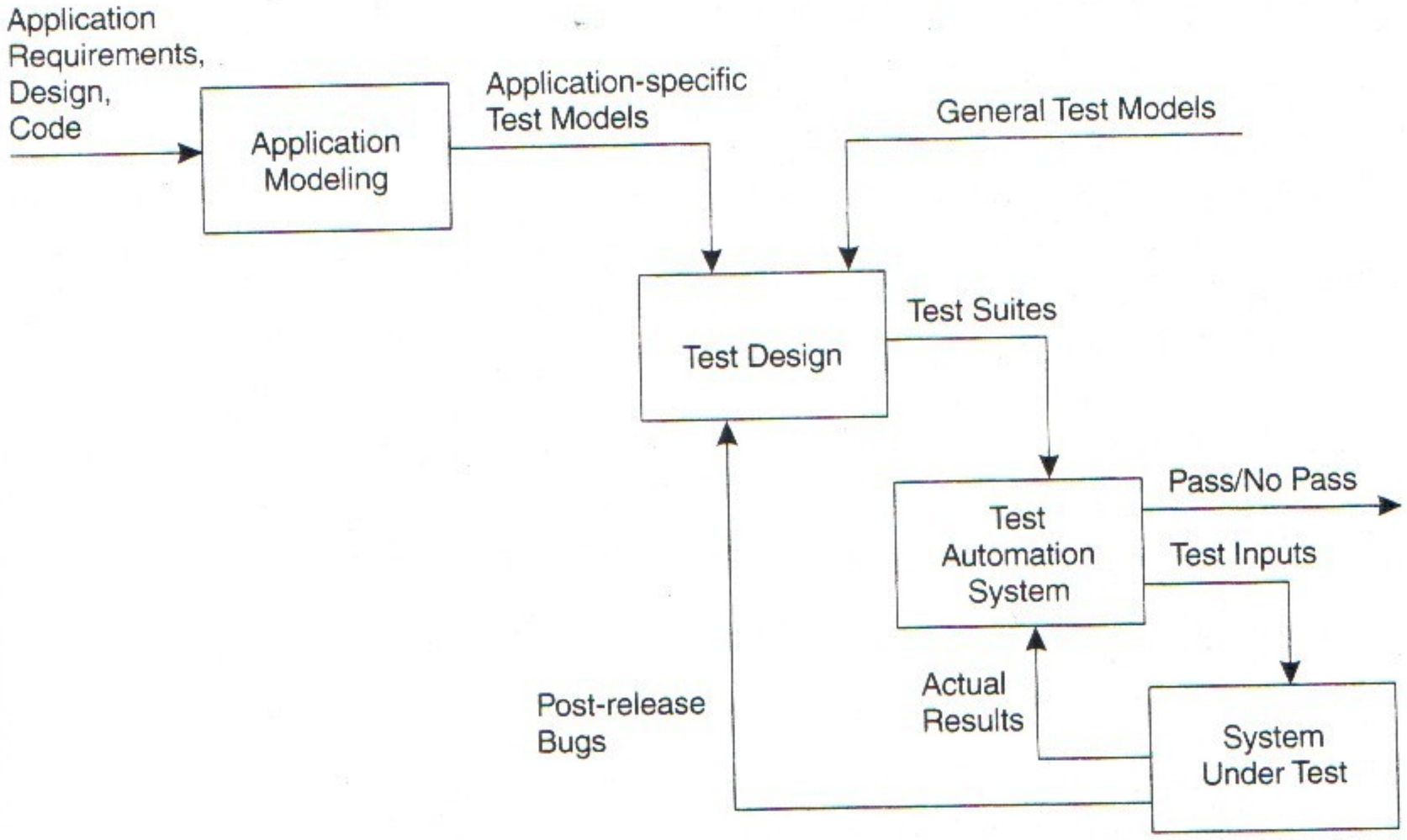

FIGURE 3.1 A systems engineering view of testing.

Test execution typically follows several steps:

1. Establish that the implementation under test is minimally operational by exercising the interfaces between its parts.
2. Execute the test suite; the result of each test is evaluated as pass or no pass.
3. Use a coverage tool to instrument the implementation under test. Rerun the test suite and evaluate the reported coverage.
4. If necessary, develop additional tests to exercise uncovered code.
5. Stop testing when the coverage goal is met and all tests pass.

Some of these steps are not always necessary or feasible, but we must nevertheless be able to run a test suite and evaluate results. Part IV provides 17 design patterns for test automation. Test design and execution are best carried out in parallel with application analysis, design, and coding. This can also follow coding, although this sequence is problematic for many reasons.

There are some things that testing is not:

- Testing is not cut-and-fit improvement of a user interface or requirements elicitation by prototyping.
- Testing is not the verification of an analysis or design model by syntax checkers or simulation.
- Testing is not the scrutiny of documentation or code by humans in inspections, reviews, or walkthroughs.
- Testing is not static analysis of code using a language translator or code checker (e.g., `lint`). Specifically, getting a clean compile is not in any sense passing a test.
- Testing is not the use of dynamic analyzers to identify memory leaks or similar problems.
- Testing is not debugging, although successful testing should lead to debugging.

All the preceding activities are important in preventing and removing software bugs. But they are not testing. At least one element of testing is missing: design, execution, or evaluation. Some writers have characterized testing as including all these and more. I reject this view. As a result, this book does not discuss other important software engineering approaches necessary to achieve high quality.

3.2 Definitions

The definitions in this section establish basic ideas needed to discuss testing. Many more definitions appear in subsequent chapters, and all definitions from the book are collected in the Glossary.

Software testing is the execution of code using combinations of input and state selected to reveal bugs.[1] Some definitions of testing include other verification and validation activities, but here testing is limited to running an implementation with input selected by test design and evaluating the response. A **component** refers to any software aggregate that has visibility in a development environment—for example, a method, a class, an object, a function, a module, an executable, a task, a utility subsystem, an application subsystem. Compo-

1. Several comprehensive introductions to software testing are available (see the Bibliographic Notes section). This section is limited to definitions that make the usage clear and support the subsequent chapters. Terms follow applicable IEEE/ANSI standards: IEEE 610.12, IEEE 829, and IEEE 982.1, unless otherwise stated.

nents include executable software entities supplied with an application programmer interface. See Section 12.1.1, What Is a Subsystem?

The **scope** of a test is the collection of software components to be verified. Since tests must be run on an executable software entity, scope is typically defined to correspond to some executable component or system of components. The code being tested is called the **implementation under test** (IUT), **method under test** (MUT), **object under test** (OUT), **class under test** (CUT), **component under test** (CUT), **system under test** (SUT), and so on. Software testing is typically categorized by the scope of the IUT and test design approach. Scope is traditionally designated as unit, integration, or system (see Figure 3.2).

The scope of a **unit test** typically comprises a relatively small executable. In object-oriented programming, an object of a class is the smallest executable unit, but test messages must be sent to a method, so we can speak of method scope testing. A test unit may consist of a class, several related classes (a cluster), or an executable binary file. Typically, it is a cluster of interdependent classes. Test design at class, cluster, and component scope is discussed in Chapters 10, 11, and 12.

The scope of an **integration test** is a complete system or subsystem of software and hardware units. The units included are physically dependent or must cooperate to meet some requirement. Integration testing exercises interfaces among units within the specified scope to demonstrate that the units are collectively operable. For example, testing the use cases implemented by a subsystem cannot begin until interprocess communication among executable processes (each composed of many objects) works well enough to run tests based on these use cases. Integration testing begins early in the programming of object-oriented software because a single class is typically composed of objects of other classes and inherits features from superclasses. Integration testing is discussed in Chapter 13.

The scope of a **system test** is a complete integrated application. Tests focus on capabilities and characteristics that are present only with the entire system. The boundary of the system under test typically excludes the virtual machine that supports it and other application systems to which it has direct or indirect interfaces. System scope tests may be categorized by the kind of conformance they seek to establish: functional (input/output), performance (response time and resource utilization), stress or load (response under maximum or overload). System testing is discussed in Chapter 14.

Testing is **fault-directed** when the intent is to reveal faults through failures; it is **conformance-directed** when the intent is to demonstrate conformance to required capabilities. These goals are not mutually exclusive. Many test design

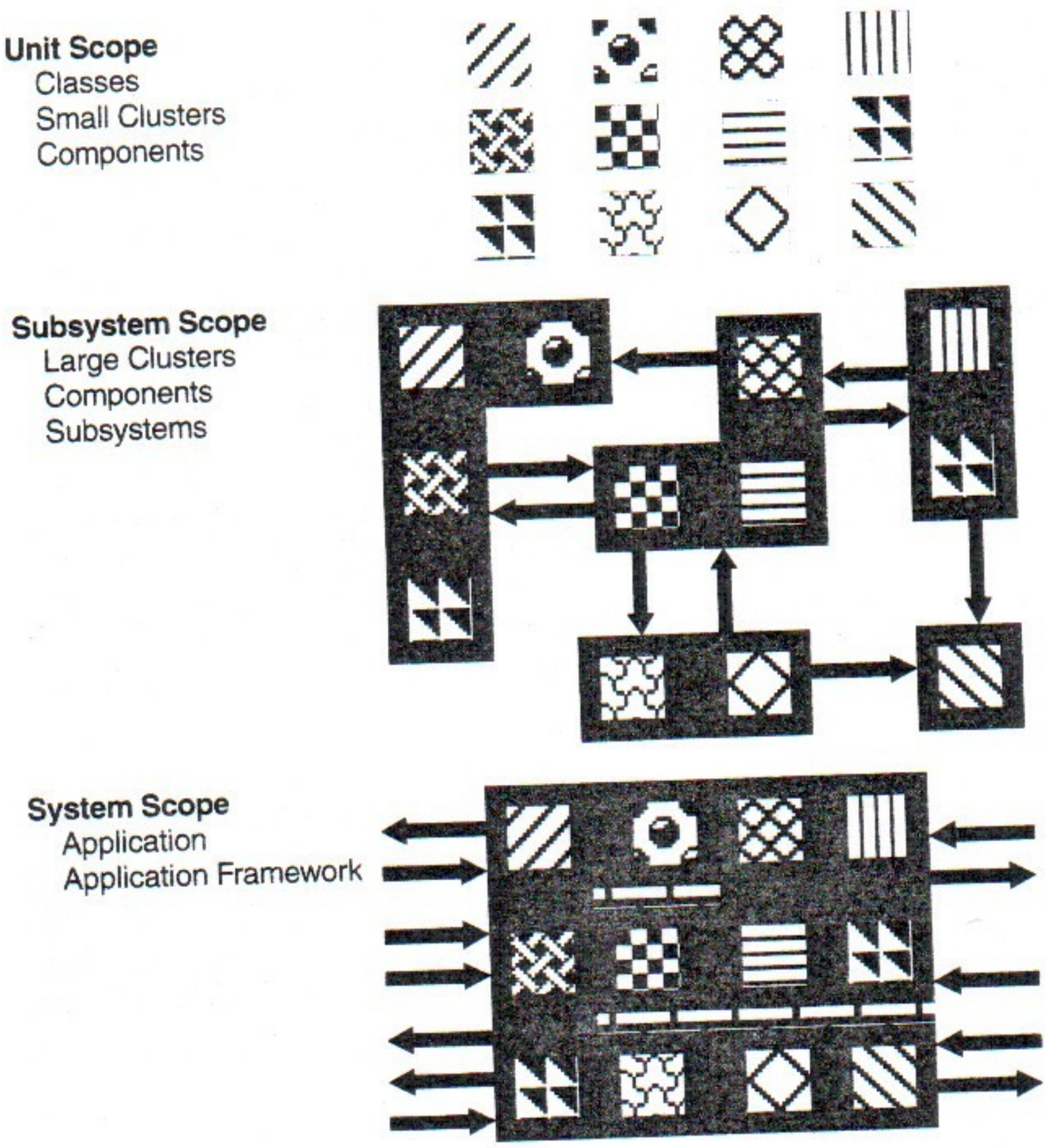

FIGURE 3.2 Relationship of unit, integration, and system tests.

techniques achieve both. Unit and integration testing are typically more fault-directed. System testing is typically more conformance-directed.

Confidence is the subjective assessment of the likelihood of unrevealed bugs in an implementation. Here, "confidence" is used in the everyday sense of a qualitative judgment based on available information. Testing provides information that can increase or decrease our confidence. When a responsibility test suite passes and achieves appropriate coverage, we may have increased confidence that the IUT will perform without failure. Confidence can be quantified when testing is conducted under certain controlled conditions [Hamlet+90]. An estimated **failure rate** is the probability that failure will occur after a certain period of usage [Friedman+95, Lyu+96, Musa 98]. Quantitative techniques for

establishing statistical confidence have been developed that are applicable to object-oriented systems; these techniques are not discussed here. This book uses the qualitative sense of confidence, unless noted otherwise.

A **test case** specifies the pretest state of the IUT and its environment, the test inputs or conditions, and the expected result. The **expected result** specifies what the IUT should produce from the test inputs. This specification includes messages generated by the IUT, exceptions, returned values, and resultant state of the IUT and its environment. Test cases may also specify initial and resulting conditions for other objects that constitute the IUT and its environment. An **oracle** is a means to produce expected results. Some automated oracles can also make a pass/no pass evaluation. Oracles are the subject of Chapter 18.

A **test point** is a specific value for test case input and state variables. A test point, which may be used in many test cases, is selected from a domain. A **domain** is a set of values that input or state variables of the IUT may take.[2] **Domain analysis** places constraints on input, state, and output values to select test values. A **subdomain** is a subset of a domain.

Some well-known heuristics for test point selection include equivalence classes, boundary value analysis, and special values testing [Myers 79]. An **equivalence class** is a set of input values such that if any value is processed correctly (incorrectly), then it is assumed that all other values will be processed correctly (incorrectly). An equivalence class does not refer to "class" as a programming language construct from which we may instantiate objects. Tests with boundary and special values are based on the assumptions that bugs are likely when input or state values are at or very near to a minimum or maximum. These techniques are examples of a general strategy called **partition testing**. A **partition** divides the input space into groups, which are hoped to have the property that any member of the group will cause a failure, if a bug exists in the code related to that partition. Although partition testing approaches have some practical advantages, the selection of partitions has no necessary relationship to the conditions necessary to reach a bug—in essence, they are guesses [Hamlet+90]. Section 10.2.4, Domain Testing Models, explains the improvements that domain analysis makes on heuristic input selection.

---

2. The testing usage of *domain* derives from the mathematical definition of domain: the set of inputs accepted by a function. It is not to be confused with: (1) an "application domain," which refers to a general class of user problems (e.g., international banking, avionics, word processing, and so on) [Kean 97], (2) a "problem domain," which is any collection of ideas related by some human purpose, (3) the domain component of an IP address, or (4) the value set of an attribute in database theory [Date 82]. The use of "domain" in the testing sense predates these definitions [Howden 76]. The UML definition of domain includes both definitions (1) and (2).

A **test suite** is a collection of test cases, typically related by a testing goal or implementation dependency. A **test run** is the execution (with results) of a test suite(s). A **test plan** is a document prepared for human use that explains a testing approach: the work plan, general procedures, explanation of the test design, and so on. See Section 9.3, Documenting Test Cases, Suites, and Plans.

The IUT produces **actual results** when a test case is applied to it. These results include messages generated by the IUT, exceptions, returned values, and the resultant state of the IUT and its environment. A test whose actual results are the same as the expected results is said to **pass**; otherwise, it is a **no pass**. A no pass test reveals a bug and is therefore a successful test, even though the actual results are typically an application failure.

A **test driver** is a class or utility program that applies test cases to an IUT. A **stub** is a partial, temporary implementation of a component. It may serve as a placeholder for an incomplete component or implement testing support code. A **test message** is the code in a test driver that is sent to a method of the object under test. Two or more test messages that follow one another make up a **test sequence**. A **test script** is a program written in a procedural script language (usually interpreted) that executes a test suite(s). A **test harness** is a system of test drivers and other tools to support test execution. Chapter 19 discusses stubs, test cases, and test drivers.

A **failure** is the manifested inability of a system or component to perform a required function within specified limits. It is evidenced by incorrect output, abnormal termination, or unmet time and space constraints. A software **fault** is missing or incorrect code. When the executable code (translated from faulty statements) is executed, it may result in a failure. An **error** is a human action that produces a software fault. Errors, faults, and failures do not occur in one-to-one relation. That is, many different failures can result from a single fault, and the same failure can be caused by different faults. Similarly, a single error may lead to many faults. These categories are generally useful, but somewhat ambiguous. Readers interested in greater precision may find the orthogonal defect classification model useful [Chillarege 96].

An **abend** is an abrupt termination of a program and is synonymous with "task abort," "application crash," or "system crash."[3] An abend often produces a memory dump—for example, the "blue screen of death" generated by Microsoft operating systems.

---

3. *Abend* is IBM jargon for abnormal end. It is pronounced "ab-end," to rhyme with "the end."

An **omission** is a required capability that is not present in an implementation. For purposes of testing, an omission must be defined with respect to required capabilities. Testing can reveal omitted capabilities. However, systems are often disappointing, dangerous, or deficient because important requirements have not been identified and therefore cannot be implemented. Omitted requirements are a serious problem, but not one that testing should address. Discovering omitted requirements during testing is possible, but only if a tester is lucky enough to notice that something is odd or absent. Better ways exist to prevent omitted requirements [Gause+89, Thayer+90, Leveson 95].

A **surprise** is code that does not support any required capability. Surprises may be either benign or malignant. The Venn Diagram in Figure 3.3 shows how requirements and an implementation relate to faults, omissions, and surprises. Object-oriented applications often contain many benign surprises. Large-grained reusable components, for example, may implement capabilities that are irrelevant for a specific application, but that cannot be removed. This irrelevant code becomes a surprise in the reusing application. If the unused reused code can be safely deactivated or avoided, this inclusion is usually not a problem. As systems evolve, however, surprises can become bugs.

A **bug** is an error or a fault. The first recorded usage of "bug" to describe a problem in computing was made by Grace Hopper in 1945 when a moth became lodged in a computer relay, causing the program to halt [Hopper 81]. Figure 3.4 shows a photograph of this moth taped into the log book with

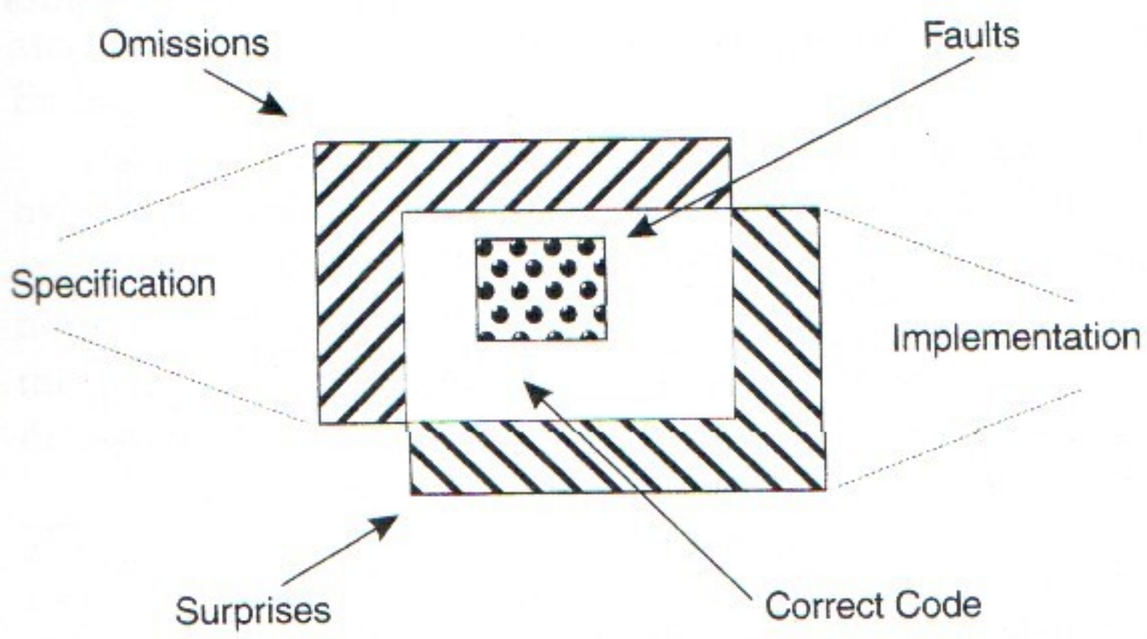

FIGURE 3.3 Faults, omissions, and surprises.

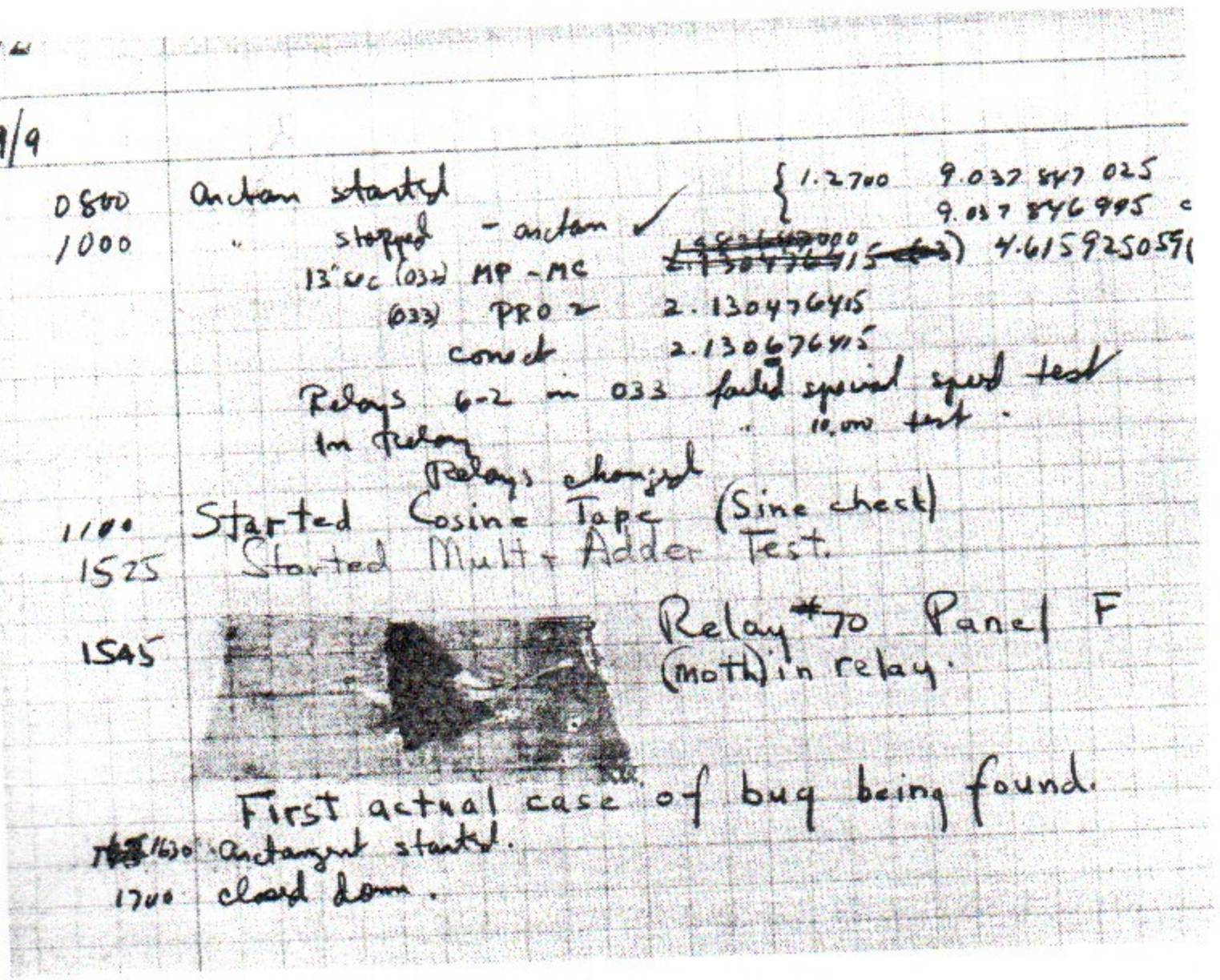

FIGURE 3.4 Hopper's bug.

Hopper's ironic note: "First actual case of bug being found." Dijkstra argues that "bug" is actually a corrupting euphemism:

> We could, for instance, begin with cleaning up our language by no longer calling a bug "a bug" but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, viz., with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it is a disguise that the error is the programmer's own creation. The nice thing of this simple change of vocabulary is that it has such a profound effect. While, before, a program with only one bug used to be "almost correct," afterwards a program with an error is just "wrong" . . . [Dijkstra 89, 1403].

Similarly, others deprecate "bug" because it connotes a malevolent being like a gremlin or leprechaun, who sabotages the developer's efforts. However, "bug" is widely used. I use "bug" unless the finer distinction of error or fault

is needed. I do not think this usage will lead to irresponsible software development practices among professionals. Developers who believe that bugs result from evil forces beyond their control or who rely on small shades of meaning as a fig leaf for incompetence are beyond hope anyway.

Debugging is the work required to diagnose and correct a bug. *Testing is not debugging. Debugging is not testing.* Debugging typically occurs after a failure has been observed or when a developer identifies a fault by inspection or automatic code analysis. It includes the analysis and experimentation necessary to isolate and diagnose the cause of a failure, the programming to correct the bug, and testing to verify that the change has removed the bug.[4] The observed failure may result from testing (which is desirable) or the intended use of a system (which is undesirable). In contrast, testing is the process of devising and executing a test suite that attempts to cause failures, increasing our confidence that failures are unlikely.

Because the number of possible tests is infinite for practical purposes, rational testing must be based on a **fault model**. This is an assumption about where faults are likely to be found. An **interesting test case** is one that has a good chance of revealing a fault. Fault models for object-oriented programming are discussed in Chapter 4. Each test design pattern has an explicit fault model.

A **test strategy** is an algorithm or heuristic to create test cases from a representation, an implementation, or a test model. A **test model** represents relationships among elements of a representation or implementation. It is typically based on a fault model. **Test design** produces a suite of test cases using a test strategy. Test design is concerned with three problems: identification of interesting test points, placing these test points into a test sequence, and defining the expected result for each test point in the sequence. **Test effectiveness** is the relative ability of testing strategy to find bugs. **Test efficiency** is the relative cost of finding a bug.

Strategies for test design are responsibility-based, implementation-based, hybrid, or fault-based. **Responsibility-based test design** uses specified or expected responsibilities of a unit, subsystem, or system to design tests. The notion of responsibility is discussed in Chapter 9. For example, a test case for the trigonometric function $\tan(x)$ could be prepared from its mathematical definition, whatever the algorithm design or implementation language. It

4. Considerations for object-oriented debugging are developed in [Purchase+91]. Smalltalk-specific techniques are presented in [Rochat+93, Hinke+93], and C++ techniques in [Thielen 92, Davis 94, Spuler 94, Ball 95]. A survey of debugging research for procedural code appears in [Agarwal+89]. A general debugging strategy for IBM mainframe code can be found in [Binder 85].

is synonymous with "specification-oriented," "behavioral," "functional," or "black box" test design.

Behavioral testing is the generally preferred term for testing that is not based on implementation-specific information [Beizer 95]. In the context of object-oriented software, however, *behavior* usually refers to a sequence of responses that an object may produce or, more generally, to its "dynamic model." Black box testing usually means that an external view of the IUT is analyzed to develop tests, but it can also describe tests derived from analysis of implementation interfaces or imputed class functionality. To avoid confusion and to make the basis for testing clear, I characterize testing as responsibility-based or implementation-based.

Implementation-based test design relies on source code analysis to develop test cases. It is synonymous with "structural," "glass box," "clear box," or "white box" test design. The term "white box testing" was probably chosen to suggest the opposite of black box testing. As a white box is just as visually opaque as a black box, clear box or glass box is occasionally used as a rubric for implementation-based testing.

An approach is said to be **formal** if its fault-revealing capability has been established by mathematical analysis or **heuristic** if it relies on expert judgment or experience to select test cases. **Hybrid test design** blends responsibility-based and implementation-based test design and is sometimes called gray box testing. **Fault-based testing** purposely introduces faults into code (mutation) to see if these faults are revealed by a test suite [Morell+92, Friedman+95]. Although not widely used, this approach has been studied in detail. This book does not present any application of fault-based testing. General test design strategies are the subject of Part II, Models. Scope and structure specific test design is the subject of Part III, Patterns.

An **exhaustive test suite** requires that every possible value and sequence of inputs be applied in every possible state of the system under test, thereby exercising every possible execution path. This approach results in an astronomical number of test cases, even with trivial programs. Exhaustive testing is a practical impossibility. Software testing is therefore necessarily concerned with small subsets of the exhaustive test suite.

The completeness of a test suite with respect to a particular test case design method is measured by coverage. **Coverage** is the percentage of elements required by a test strategy that have been exercised by a given test suite. Many coverage models have been proposed and studied [Ntafos 88, Beizer 90, Kaner +93, Roper 94, Zhu+97]. For example, **statement coverage** is the percentage of all source code statements executed at least once by a test suite. The coding con-

structs that make up statements differ for various programming languages. In C++, Java, and Objective-C, an expression terminated by a semicolon is a statement. A **predicate** statement results in a conditional transfer of control (if-then, loop control, case, and so on). A **branch** is one of several statements that can immediately follow a predicate statement. **Branch coverage** requires that every branch be exercised at least once. When the term "coverage" is used without quantification, 100 percent is usually understood; for example, "This test suite achieves branch coverage for member function *x*" means that 100 percent branch coverage has been achieved. Coverage of object-oriented code is discussed in Section 10.2.2, Implementation-based Test Models.

3.3 The Limits of Testing

We know from experience that any nontrivial software system will have bugs. We cannot know, in advance, exactly what combination of execution sequence, state, and input will cause a failure (excluding mutation, sabotage, and any other willfully incorrect coding). Clearly, passing a single test is not enough to show the absence of bugs. Output may depend on an internal state and typically only a small part of the code in an implementation is used for any given input. So, just because `f(x)` returns a correct result when `x = 123`, we cannot assume that it does so for all values of `x` or even for the *same* value of `x` when run in different circumstances (this consideration is the nub of the infamous Y2K problem). Even passing many tests for many different values of `x` does not *necessarily* mean that `f(x)` will perform correctly for the values that we do not test.

3.3.1 The Input/State Space

The number of input and output combinations for trivial programs is surprisingly large. It is astronomical for typical programs and beyond comprehension for typical systems. Consider the class `Triangle` described in Chapter 1. If we limit points to integers between 1 and 10, there are 10^4 possible ways to draw a line. Considering three lines at a time, we have $10^4 \times 10^4 \times 10^4 = 10^{12}$ possible inputs of three lines, including all invalid combinations.[5] Suppose you

5. A line rendering is defined by two end points, each consisting of an x and y coordinate. If we limit the x and y axis to values from 1 to 10, we get $10 \times 10 = 100 = 10^2$ possible x, y coordinates at each end. Including lines of no length (both end points are the same), we have $10^2 \times 10^2 = 10^4$ possible ways to draw a line.

are a *very* fast and tireless tester: you can run and check 1000 line tests per second, 24 hours per day, 365 days per year. You could test every possible input in $10^{12}/10^3 = 10^9$ seconds. At 3.154×10^7 seconds per year, this effort would take $10^9/3.154 \times 10^7 = 3.171 \times 10^2$, or a little more than 317 years. Thus if you had started testing around 1683 and assuming no system crashes and debugging, you'd be nearly finished now.

This case is actually a gross oversimplification. Assuming a typical raster display of 786,432 pixels (1024×768), a line can be drawn in $786{,}432^2$ ways (all possible pixel pairs). With three lines, $786{,}432^6$ possible test cases exist, (roughly 2.36574×10^{35}). If you ran tests nonstop, 24 hours per day, you'd need roughly the estimated age of the universe to test every possible three-line combination. Clearly, we can never test all inputs, states, or outputs.

3.3.2 Execution Sequences

Branching and dynamic binding result in a very large number of unique execution sequences for any given program. Simple iteration increases the number of possible sequences to astronomical proportions. Consider the following code fragment:

```
for ( int i = 0; i < n; ++i) {
    if ( a.get(i) == b.get(i) )
            x [ i ] = x [ i ] + 100;
    else
            x [ i ] = x [ i ] / 2;
}
```

Figure 3.5 models statement execution sequences of this loop as a flow graph (the details of flow graphs and path modeling are discussed in Section 10.2.2). If we count entry–exit paths without regarding iteration, there are only three: (1) the loop is skipped, (2) the loop is entered and the first branch is taken, and (3) the loop is entered and the second branch is taken. In terms of the nodes in the flow graph:

1. Loop Header, exit
2. Loop Header, Conditional, +100
3. Loop Header, Conditional, /2

Each possible iteration of the loop, however, doubles the number of possible paths. For example, with two iterations, there are the following five paths:

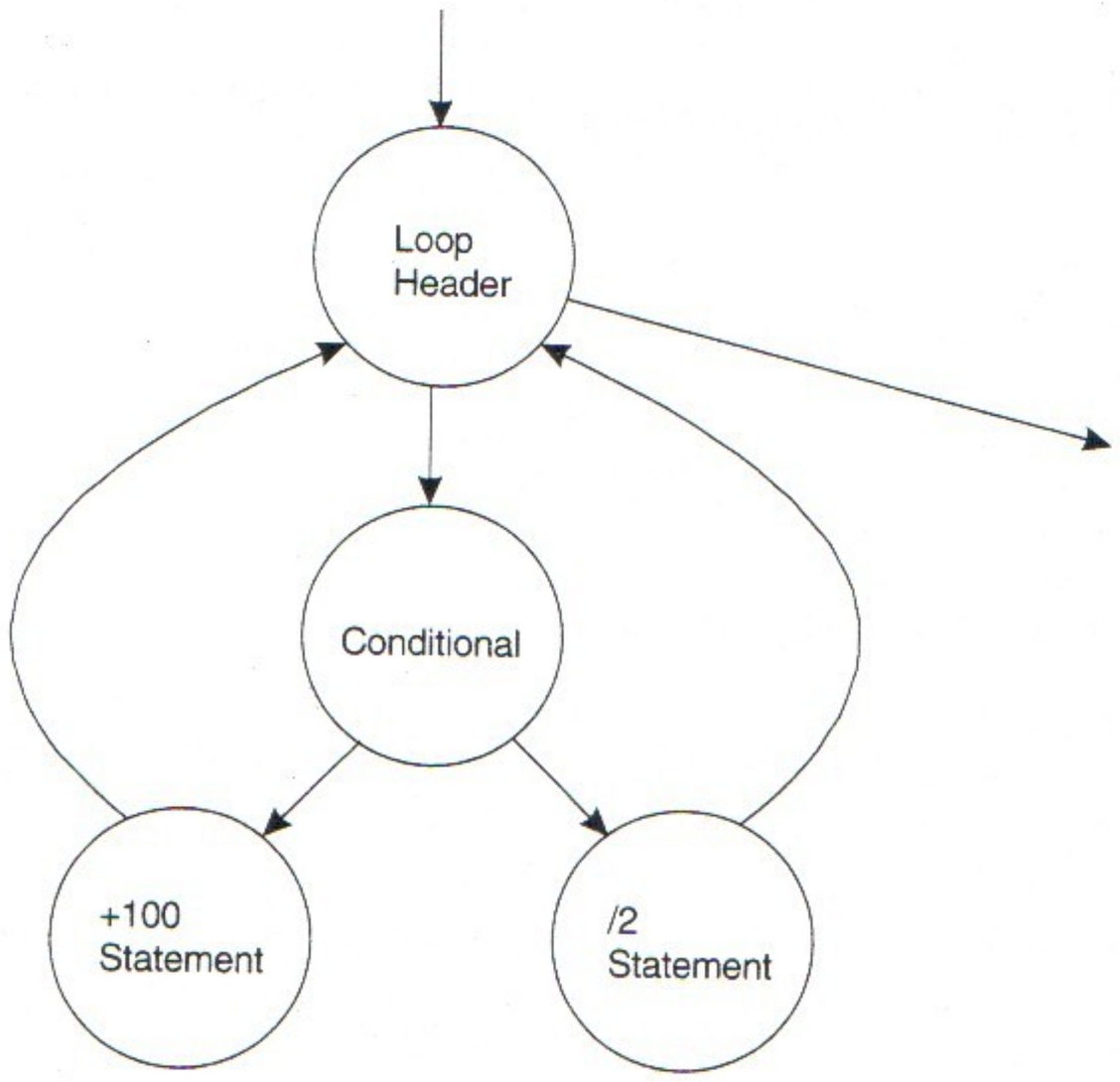

FIGURE 3.5 Flow graph for a simple loop.

1. Loop Header, exit
2. Loop Header, Conditional, +100, Conditional, +100
3. Loop Header, Conditional, +100, Conditional, /2
4. Loop Header, Conditional, /2, Conditional, +100
5. Loop Header, Conditional, /2, Conditional, /2

That is, the number of possible paths is $2^n + 1$ (for $n > 0$). If you had started testing just after the Big Bang (about 18 billion years ago) and all went well, you'd be working on $n = 36$ today.[6] Table 3.1 shows some of the arithmetic. Any day now . . .

---

6. The Big Bang is estimated to have occurred between 12 and 18 billion years ago. Assuming the largest value, 1.8×10^9 years, and testing at 1×10^3 paths/second, 3.154×10^7 seconds per year, we have 1.8×10^9 years $\times 3.154 \times 10^7 = 5.6772 \times 10^{16}$ seconds since the Big Bang. If testing started just after the Big Bang and ran at 10^3 paths per second, we could exercise $5.6772 \times 10^{16} \times 10^3$ paths, or 5.6772×10^{19} paths. At what loop value, n, would we run out of time? Solving for n, $2^n = 5.6772 \times 10^{19}$. Reducing gives $\log(2^n) = \log(5.6772 \times 10^{19})$. Then $n \log(2) = 19 \log(5.6772) = n\ 0.30103 = 19 \times 0.754134$. So $n = 3.321928 \times 19 \times 0.754134 = 35.89559$.

TABLE 3.1 Paths Due to Iteration

Number of Iterations	Number of Paths
1	3
2	5
3	9
10	1025
20	1,048,577
60	1,152,921,504,606,847,200

3.3.3 Fault Sensitivity and Coincidental Correctness

The ability of code to hide faults from a test suite is called its **fault sensitivity** [Friedman 95]. From a testing perspective, the best kind of bug is one that causes a failure every time it executes. If all bugs were so cooperative, we could guarantee bug-free code by designing a test suite that executed every line of code at least once. As we will see in Chapter 10, developing a test suite that will exercise each line of code at least once is relatively easy. Unfortunately, most bugs will not cooperate: they hide (no failure is produced) even when buggy code executes.

Coincidental correctness obtains when buggy code can produce correct results for some inputs. Suppose `x + x` is incorrectly coded instead of `x * x`. When `x` is 2 in a test suite, this code hides the bug: it produces a correct result from buggy code. Although only a little more testing is required to reveal this bug, simple errors can result in very pernicious fault hiding. Consider the following code fragment (adapted from Friedman 95, 34):

```
int scale(int j) {
    j = j - 1;        // should be j = j + 1
    j = j/30000;
    return j;
}
```

Assuming 65,536 (2^{16}) possible values for `j`, only the following 40 values will produce an incorrect result: –30000, –29999, –27000, –26999, –24000, –23999, –21000, –20999, –18000, –17999, –15000, –14999, –12000, –11999, –9000, –8999, –6000, –5999, –3000, –2999, 2999, 3000, 5999, 6000, 8999, 9000, 11999, 12000, 14999, 15000, 17999, 18000, 20999, 21000, 23999, 24000, 26999, 27000, 29999, 30000, none of which are on a boundary of `j`. Thus, we could do near-exhaustive testing of this code, covering 99.939 percent of all input values and not reveal the bug.

The effects of inheritance and polymorphism provide many opportunities for coincidental correctness. Consider the following Java fragment:

```
public class Account extends Object {
    protected Date lastTxDate, today;
    // . . .
    int quartersSinceLastTx( ) {
            return (90 / daysSinceLastTx( ));
    }

    int daysSinceLastTx( ){
            return (today.day() - lastTxDay.txDate + 1);

            // Correct - today's transactions return 1 day elapsed
    }
}

public class TimeDepositAccount extends Account{
    // . . .
    int daysSinceLastTx( )  {
            return (today.day( ) - lastTxDate.txDay);

            // Incorrect - today's transactions return 0 days
elapsed
    }
```

This code is correct for an `Account` object and can be coincidentally correct for a `TimeDepositAccount` object. No faults would be revealed when methods `quartersSinceLastTx( )` and `daysSinceLastTx( )` are tested on an `Account` object, even when we include a test case where the last transaction has occurred on the current day. When the last transaction occurs on the current day, however, a `TimeDepositAccount` object will fail with divide by zero exception, but an `Account` object will not. Bugs of this kind are more likely with deeper and wider class hierarchies. Furthermore, because `quartersSinceLastTx()` is inherited with no change, we might be tempted to skip retesting it when we develop the subclass.

Much of the test design presented in this book is motivated by the wide span of implicit dependencies in object-oriented programming languages. The test design strategies in Part III provide techniques to exercise a class systematically and shine a light into the deep crevasse of coincidental correctness resulting from polymorphism and inheritance.

3.3.4 Absolute Limitations

The kinds of bugs that testing can reveal are limited by some fundamental properties of software systems:

- In advocating proof of correctness, Dijkstra observed that "Program testing can be used to show the presence of defects, but never their absence!" [Dahl+72]. Proof of correctness is equivalent to exhaustive testing. A proof shows by analysis whether a program must produce correct results for all possible inputs. Exhaustive testing of a correct program will produce a pass on every input. As the preceding examples illustrate, however, exhaustive testing is usually impossible.[7] Because an exhaustive test cannot always be achieved, testing cannot always prove correctness (although it may be able to sometimes).
- Automating certain aspects of test design is provably impossible. No system can be completely self-descriptive, so situations arise where the system cannot produce an answer about itself.[8] While this limitation has no significance for practical test design, it may be a consideration for advanced test automation. The case for this conclusion has been well made elsewhere [Manna+78] and cannot be easily summarized, so it will not be discussed here.
- Testing must use requirements (that is, some model of required capabilities) as a point of reference. It cannot directly verify requirements. Spurious tests may be produced if requirements are incorrect or incomplete.
- Implementation-based testing cannot reveal omissions as missing code cannot be tested.
- We can never be sure that a testing system is correct. That is, bugs in test design, an oracle, or test drivers can produce spurious test results [Manna+78].
- Devising an oracle is difficult and sometimes impossible [Weyuker 82]. Without trusted expected results to compare with actual results, pass/no pass evaluation is dubious. This problem and some solutions are discussed in Chapter 18.

3.4 What Can Testing Accomplish?

Testing is complex, critical, and challenging. No "right way" exists that works every time and no "silver bullet" can solve every test design or quality problem.

7. Computer scientists would say exhaustive testing is **intractable**: while it may be possible in trivial cases, even moderately complex programs require prohibitively large quantities of storage or computation.

8. Computer scientists call such problems **undecidable**: it can be shown that no algorithm exists that can perform the required computation in every possible case.

The primary role of software testing is to reveal bugs that would be too costly or impossible to find with other verification and validation techniques. A secondary purpose is to show that the system under test complies with its stated requirements for a given test suite.

Testing is not a substitute for preventing faults by good software engineering. Instead, it should focus on areas where it is most effective and be complemented with other known-good software engineering practices. It is not the technique of choice for preventing or identifying all kinds of bugs. For example, testing rarely identifies missing requirements, but it can reveal interactions and special cases that might cause a system to crash. It does not make sense to try to "test quality into a system." Testing should not be the first step in bug prevention, nor should it be used as a final toxic waste filter. Figure 3.6 lists bug categories for which testing is the best technique.

How well or poorly testing activities are harmonized with a software process significantly affects testing effectiveness. Testing is most effective when test development begins at the outset of a project. Early consideration of testability, early test design, and implementation of tests as soon as possible provide a powerful bug prevention approach. This effect can improve application design by forcing closer consideration of the implications of requirements and specifications. Early decisions (or their absence) about architecture, detail design, and coding practices can make testing easier and cheaper, or harder and dearer.

Finally, from a software product perspective, effective testing is necessary to produce reliable, safe, and successful systems. Although quantitative data on testing and object-oriented systems are sparse, several reports show that testing can lead to very high quality. After systematic testing at class and cluster scope was adopted, system test defects in an Eiffel and C++ telecommunication application were reduced by a factor of 70. The total development effort was cut by half [Murphy+94]. Other reports indicate that effective testing in object-oriented development can contribute to very low defect rates. Their results can be compared by using function points to normalize system size.[9]

- The test process used to develop the ICpack 201 library (50 classes and 1200 Objective-C methods) was a significant factor in achieving 0.0121

9. The number of major post-release bugs and size of the system in lines of code are given in each report. The system size in function points was estimated by using the language to function point conversion factor given in [Jones 97b]. Table 3.2 shows the reported values, factors, and normalized bug rates. Details of function point models for object-oriented systems are developed in [Caldiera+98].

Type of Bug	General Cause	Applicable Validation and Verification		
		Early Validation	Static Verification	Testing
Lacks usability	Poor user interface design	✓		
Missing capability	Elicitation error	✓		
	Specification omission		✓	✓
Incorrect capability	Elicitation error	✓		✓
	Specification error	✓	✓	✓
Side effects and unanticipated, undesirable feature interaction	Elicitation error		✓	✓
	Specification error		✓	✓
	Programming error		✓	✓
	Configuration/integration error			✓
Inadequate performance, real-time deadline failure, synchronization deadlock, livelock, and so on	Elicitation error	✓		✓
	Specification error	✓	✓	✓
	Inappropriate target		✓	✓
	Inefficient algorithm		✓	✓
	Inefficient programming		✓	✓
	Infeasible problem		✓	✓
	Programming error		✓	✓
	Configuration/integration error			✓
Incorrect output	Specification error		✓	✓
	Programming error		✓	✓
	Configuration/integration error		✓	✓
Abend	Programming error		✓	✓
	Configuration/integration error			✓

Key: ✓ = Useful, ☐ = Typically most effective

FIGURE 3.6 Bugs that testing can catch.

TABLE 3.2 Reported Postrelease Bug Rates after Testing Some OO Systems

Language	LOC	LOC/FP Factor [Jones 97b]	Estimated Function Points	Major Post-release Bugs	Bugs/FP	Source
C++	75,000	55	1364	7	0.0051	[Boisvert 97]
Objective-C	12,000	29	414	5	0.0121	[Love 92]

delivered defects per function point. Testing was the primary verification technique [Love 92].

- Systematic testing at class, cluster, and system scope of a 75 KLOC C++ cellular support application resulted in 0.0051 delivered defects per function point. No other verification techniques were used [Boisvert 97].

Even allowing a factor of 10 for the estimating error, these rates compare favorably with benchmarks for "best in class" development organizations, which Jones reports as an "average less than 0.025 user-reported defects per function point" in the first year after release [Jones 97a, 44]. Jones' rate includes all defects and presumably would be lower if only major defects were considered. These reports show that effective object-oriented testing can make a major contribution to achieving world-class quality.

3.5 Bibliographic Notes

Hundreds of books and thousands of papers have been published on testing. The principles presented in Glenford Myers' *Software Reliability* [Myers 76] and *The Art of Software Testing* [Myers 79] are more than 20 years old but continue to be applicable. Both are readable and practical guides to software testing and quality. They are relevant if you mentally substitute present-day equivalents for the occasional references to obsolete technology. In a technology that churns as much as software, such durability is rare. Chapters 1 and 2 of *Software Testing Techniques* [Beizer 90] explain the test design problem in down-to-earth terms without trivializing. The presentation of test design technique that follows is unmatched in synthesizing research and practice (through 1989) and maintaining a relentless focus on test effectiveness. Kaner et al. provide a comprehensive discussion of the practical and procedural aspects of

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.