As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

ADVICE

*Regression testing is an important strategy for reducing "side effects." Run regression tests every time a major change is made to the software (including the integration of new components).*

**Regression testing.** Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. *Capture/playback tools* enable the software engineer to capture test cases and results for subsequent playback and comparison. The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

KEY POINT

Smoke testing might be characterized as a rolling integration strategy. The software is rebuilt (with new components added) and smoke tested every day.

**Smoke testing.** *Smoke testing* is an integration testing approach that is commonly used when software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show

stopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds and the entire product (in its current form) is smoke tested *daily.* The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell [MCO96] describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of benefits when it is applied on complex, time-critical software engineering projects:

- *Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- *The quality of the end-product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- *Error diagnosis and correction are simplified.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

**"Treat the daily build as the heartbeat of the project. If there's no heartbeat, the project is dead."**
**Jim McCarthy**

**Strategic options.** There has been much discussion (e.g., [BEI84]) of the relative advantages and disadvantages of top-down versus bottom-up integration testing. In general, the advantages of one strategy tend to result in disadvantages for the other strategy. The major disadvantage of the top-down approach is the need for stubs and

**WebRef**
Pointers to commentary on testing strategies can be found at www.qalinks.com.

the attendant testing difficulties that can be associated with them. Problems associated with stubs may be offset by the advantage of testing major control functions early. The major disadvantage of bottom-up integration is that "the program as an entity does not exist until the last module is added" [MYE79]. This drawback is tempered by easier test case design and a lack of stubs.

Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes called *sandwich testing*) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

**What is a "critical module" and why should we identify it?**

As integration testing is conducted, the tester should identify critical modules. A *critical module* has one or more of the following characteristics: (1) addresses several software requirements, (2) has a high level of control (resides relatively high in the program structure), (3) is complex or error prone, or (4) has definite performance requirements. Critical modules should be tested as early as possible. In addition, regression tests should focus on critical module functions.

**Integration test documentation.** An overall plan for integration of the software and a description of specific tests are documented in a *Test Specification.* This document contains a test plan, a test procedure, is a work product of the software process, and becomes part of the software configuration.

The test plan describes the overall strategy for integration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for a CAD system might be divided into the following test phases:

- User interaction (command selection, drawing creation, display representation, error processing and representation).
- Data manipulation and analysis (symbol creation, dimensioning, rotation, computation of physical properties).
- Display processing and generation (two-dimensional displays, three-dimensional displays, graphs and charts).
- Database management (access, update, integrity, performance).

Each of these phases and subphases (denoted in parentheses) delineates a broad functional category within the software and can generally be related to a specific domain within the software architecture. Therefore, program builds (groups of modules) are created to correspond to each phase. The following criteria and corresponding tests are applied for all test phases:

*Interface integrity.* Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

*Functional validity.* Tests designed to uncover functional errors are conducted.

*Information content.* Tests designed to uncover errors associated with local or global data structures are conducted.

*Performance.* Tests designed to verify performance bounds established during software design are conducted.

A schedule for integration, the development of overhead software, and related topics is also discussed as part of the test plan. Start and end dates for each phase are established and "availability windows" for unit tested modules are defined. A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort. Finally, test environment and resources are described. Unusual hardware configurations, exotic simulators, and special test tools or techniques are a few of many topics that may also be discussed.

The detailed testing procedure that is required to accomplish the test plan is described next. The order of integration and corresponding tests at each integration step are described. A listing of all test cases (annotated for subsequent reference) and expected results is also included.

A history of actual test results, problems, or peculiarities is recorded in a *Test Report* that can be appended to the *Test Specification,* if desired. Information contained in this section can be vital during software maintenance. Appropriate references and appendixes are also presented.

Like all other elements of a software configuration, the test specification format may be tailored to the local needs of a software engineering organization. It is important to note, however, that an integration strategy (contained in a test plan) and testing details (described in a test procedure) are essential ingredients and must appear.

**"The best tester isn't the one who finds the most bugs . . . the best tester is the one who gets the most bugs fixed."**

**Cem Kaner et al.**

## 13.4 Test Strategies for Object-Oriented Software

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of object-oriented software changes both testing strategy and testing tactics (Chapter 14).

### 13.4.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes. This means that each class and each instance of a class (object) packages attributes (data) and the operations (functions) that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations within the class are the smallest testable units. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

**KEY POINT**
Class testing for OO software is analogous to module testing for conventional software. It is not advisable to test operations in isolation.

We can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class. To illustrate, consider a class hierarchy in which an operation *X* is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation *X*, but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation *X* is used varies in subtle ways, it is necessary to test operation *X* in the context of each of the subclasses. This means that testing operation *X* in a standalone fashion (the conventional unit testing approach) is usually ineffective in the object-oriented context.

Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

### 13.4.2 Integration Testing in the OO Context

Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies (Section 13.3.2) have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class" [BER93].

**KEY POINT**
An important strategy for integration testing of OO software is thread-based testing. Threads are sets of classes that respond to an input or event. Use-based tests focus on classes that do not collaborate heavily with other classes.

There are two different strategies for integration testing of OO systems [BIN94]. The first, *thread-based testing,* integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* which use the independent classes, are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

The use of drivers and stubs also changes when integration testing of OO systems is conducted. Drivers can be used to test operations at the lowest level and for the testing of whole groups of classes. A driver can also be used to replace the user interface so that tests of system functionality can be conducted prior to implementation of the interface. Stubs can be used in situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented.

*Cluster testing* is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

## 13.5 Validation Testing

*Validation testing* begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between conventional and object-oriented software disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

**KEY POINT**
Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end-user.

*Validation* can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

Reasonable expectations are defined in the *Software Requirements Specification*—a document that describes all user-visible attributes of the software. The specification contains a section called *Validation Criteria.* Information contained in that section forms the basis for a validation testing approach.

### 13.5.1 Validation Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exist: (1) The function or performance characteristic conforms to specification and is accepted, or (2) a deviation from specification is uncovered and a deficiency list is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

### 13.5.2 Configuration Review

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometimes called an *audit,* is discussed in more detail in Chapter 27.

### 13.5.3 Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

**"Given enough eyeballs, all bugs are shallow (e.g., given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone)."**

**E. Raymond**

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called *alpha* and *beta testing* to uncover errors that only the end-user seems able to find.

The *alpha test* is conducted at the developer's site by end-users. The software is used in a natural setting with the developer "looking over the shoulder" of typical users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at end-user sites. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The end-user records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

**SAFEHOME**

### *Preparing for Validation*

**The scene:** Doug Miller's office, as component-level design continues and construction of certain components begins.

**The players:** Doug Miller, software engineering manager, Vinod, Jamie, Ed, and Shakira—members of the *SafeHome* software engineering team.

**The conversation:**

**Doug:** The first increment will be ready for validation in what . . . about three weeks?

**Vinod:** That's about right. Integration is going well. We're smoke testing daily, finding some bugs but nothing we can't handle. So far, so good.

**Doug:** Talk to me about validation.

**Shakira:** Well, we'll use all of the use-cases as the basis for our test design. I haven't started yet, but I'll be developing tests for all of the use-cases that I've been responsible for.

**Ed.** Same here.

[illegible] but we've got to get our act together for [illegible] for alpha and beta testing, no?

[illegible] I've been thinking that we could bring [illegible] to help us with validation. I have [illegible] and it would give us a new [illegible]

[illegible] we've got it under control.

[illegible] But an ITG gives us an [illegible] the software.

[illegible] here, Doug. I, for one, [illegible] anybody you bring in to

**Doug:** I know, I know. But if an ITG works from requirements and use-cases, not too much baby sitting will be required.

**Vinod:** I still think we've got it under control.

**Doug:** I hear you, Vinod, but I'm going to overrule on this one. Let's plan to meet with the ITG rep later this week. Get 'em started and see what they come up with.

**Vinod:** Okay, maybe it'll lighten the load a bit.

## 13.6 System Testing

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

"Like death and taxes, testing is both unpleasant and inevitable."

Ed Yourdon

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem. Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger-pointing does occur, and (4) participate in planning and design of system tests to ensure that software is adequately tested.

*System testing* is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests [BEI84] that are worthwhile for software-based systems.

### 13.6.1 Recovery Testing

Many computer-based systems must recover from faults and resume processing within a prespecified time. In some cases, a system must be *fault tolerant*; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

*Recovery testing* is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the *mean-time-to-repair* (MTTR) is evaluated to determine whether it is within acceptable limits.

### 13.6.2 Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

*Security testing* verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer [BEI84]: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

### 13.6.3 Stress Testing

Software testing steps discussed earlier in this chapter result in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

*Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause memory management problems are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to overwhelm the program.

**"If you're trying to find true system bugs and you haven't subjected your software to a real stress test, then it is high time you started."**

**Boris Beizer**

A variation of stress testing is a technique called *sensitivity testing.* In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

### 13.6.4 Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. *Performance testing* is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

SOFTWARE TOOLS

#### *Test Planning and Management*

**Objective:** These tools assist the software team in planning the testing strategy that is chosen and managing the testing process as it is conducted.

**Mechanics:** Tools in this category address test planning, test storage, management and control, requirements traceability, integration, error tracking, and report generation. Project managers use them to supplement project scheduling tools. Testers use these tools to plan testing activities and to control the flow of information as the testing process proceeds.

**Representative Tools**[2]

*OTF (Object Testing Framework)*, developed by MCG Software, Inc. (www.mcgsoft.com), provides a framework for managing suites of tests for Smalltalk objects.

*QADirector*, developed by Compuware Corp. (www.compuware.com/qacenter), provides a single point of control for managing all phases of the testing process.

*TestWorks*, developed by Software Research, Inc. (www.soft.com/Products/index.html), contains a fully integrated suite of testing tools including tools for test management and reporting.

## 13.7 The Art of Debugging

Software testing is an action that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

*Debugging* occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is an action that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

**"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."**

**Maurice Wilkes, discovers debugging, 1949**

### 13.7.1 The Debugging Process

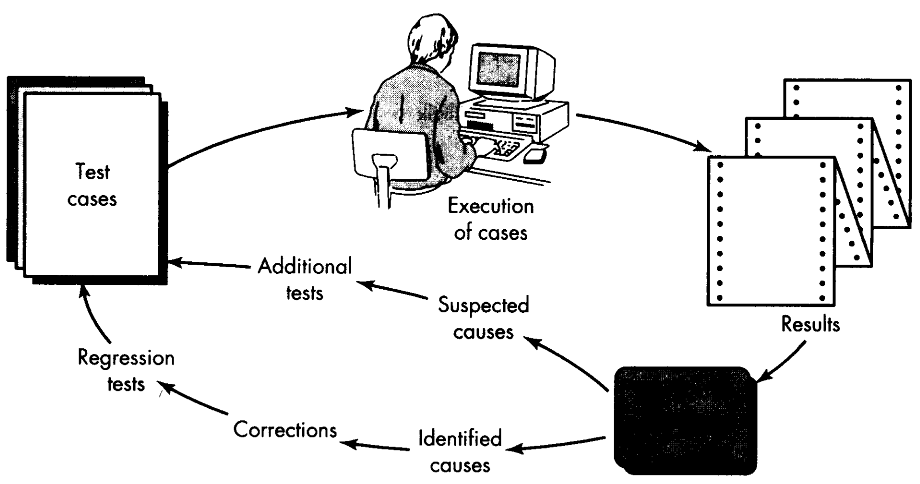
Debugging is not testing but always occurs as a consequence of testing.[3] Referring to Figure 13.7, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. Debugging attempts to match symptom with cause, thereby leading to error correction.

Debugging will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In the latter case, the person perform-

2 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

3 In making the statement, we take the broadest possible view of testing. Not only does the developer test software prior to release, but the customer/user tests software every time it is used!

FIGURE 13.7
The debugging process



ing debugging may suspect a cause, design one or more test cases to help validate that suspicion, and work toward error correction in an iterative fashion.

Why is debugging so difficult? In all likelihood, human psychology (see the next section) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

**Why is debugging so difficult?**

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 11) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors [CHE90].

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces a software developer to fix one error while at the same time introducing two more.

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?"

Brian Kernighan

## 13.7.2 Psychological Considerations

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it, and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience.

Commenting on the human aspects of debugging, Shneiderman [SHN80] states:

> Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately . . . corrected.

Although it may be difficult to "learn" debugging, a number of approaches to the problem can be proposed. We examine these in the next section.

SafeHome

**Debugging**

**The scene:** Ed's cubical as coding is conducted.

[illegible] Ed and Shakira—members of the [illegible] engineering team.

[illegible]

**[illegible] in through the entrance to** [illegible] . . . where were you at lunch time?

[illegible] working.

[illegible] miserable . . . what's the matter?

[illegible] I've been working on this [illegible] discovered it at 9:30 this morning, [illegible] I'm clueless.

[illegible] we all agreed to spend no more [illegible] stuff on our own, then we'd get [illegible]

**Ed:** Yeah, but . . .

**Shakira (walking into the** [illegible] problem?

**Ed:** It's complicated, And besides, I've been [illegible] this for, what, 5 hours? You're not going to [illegible]

**Shakira:** Indulge me . . . what's the [illegible]

(Ed explains the problem to Shakira, who [illegible] about 30 seconds without speaking.)

**Shakira (a smile gathering on her** [illegible] right there, the variable named [illegible] Shouldn't it be set to "false" before the [illegible]

(Ed stares at the screen in disbelief, [illegible] begins to bang his head gently against [illegible] Shakira, smiling broadly now, [illegible]

### 13.7.3 Debugging Strategies

Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition, and luck. Bradley [BRA85] describes the debugging approach in this way:

> Debugging is a straightforward application of the scientific method that has been developed over 2,500 years. The basis of debugging is to locate the problem's source [the cause] by binary partitioning, through working hypotheses that predict new values to be examined.
>
> Take a simple non-software example: A lamp in my house does not work. If nothing in the house works, the cause must be in the main circuit breaker or outside. I look around to see whether the neighborhood is blacked out. I plug the suspect lamp into a working socket and a working appliance into the suspect circuit. So goes the alternation of hypothesis and test.

In general, three debugging strategies have been proposed [MYE79]: (1) brute force, (2) backtracking, and (3) cause elimination. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

**"The first step in fixing a broken program is getting it to fail repeatably (on the simplest example possible)."**
**T. Duff**

**ADVICE**

*Set a time limit, say, one hour, on the time you spend trying to debug a problem on your own. After that, get help!*

**Debugging tactics.** The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

*Backtracking* is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

The third approach to debugging—*cause elimination*—is manifested by induction or deduction and introduces the concept of *binary partitioning*. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised, and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

**Automated debugging.** Each of these debugging approaches can be supplemented with debugging tools that provide semi-automated support for the software engineer as debugging strategies are attempted. Hailpern and Santhanam [HAI02] summarize the state of these tools when they note, ". . . many new approaches have been proposed and many commercial debugging environments are available. Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation." One area that has caught the imagination of the industry is the visualization of the necessary underlying programming constructs as a means to analyze a program [BAE97]. A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

SOFTWARE TOOLS

### *Debugging*

**Objective:** These tools provide automated assistance for those who must debug software problems. The intent is to provide insight that may be difficult to obtain if approaching the debugging process manually.

**Mechanics:** Most debugging tools are programming language and environment specific. Debugging features assist in the diagnosis of errors that are found.

**Representative Tools**[4]

*Jprobe ThreadAnalyzer,* developed by Sitraka (www.sitraka.com), helps in the evaluation of thread problems—deadlocks, stalls, and race conditions that can pose serious hazards to application performance in Java apps.

*C++Test,* developed by Parasoft (www.parasoft.com), is a unit testing tool that supports a full range of tests on C and C++ code.

*CodeMedic,* developed by NewPlanet Software (www.newplanetsoftware.com/medic/), provides a graphical interface for the standard UNIX debugger, *gdb,* and implements its most important features. *gdb* currently supports C/C++, Java, PalmOS, various embedded systems, assembly language, FORTRAN, and Modula-2.

*BugCollector Pro,* developed by Nesbitt Software Corp. (www.nesbitt.com/), implements a multiuser database that assists a software team in keeping track of reported bugs and other maintenance requests and managing debugging workflow.

*GNATS,* a freeware application (www.gnu.org/software/gnats/), is a set of tools for tracking bug reports.

**The people factor.** Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! A fresh viewpoint, unclouded by hours of frustration, can do wonders.[5] A final maxim for debugging might be: When all else fails, get help!

4 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

5 The concept of pair programming (recommended as part of the Extreme Programming process model discussed in Chapter 4) provides a mechanism for "debugging" as the software is designed and coded.

### 13.7.4 Correcting the Error

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck [VAN89] suggests three simple questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

**When I correct an error, what questions should I ask myself?**

1. *Is the cause of the bug reproduced in another part of the program?* In many situations, a program error is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.
2. *What "next bug" might be introduced by the fix that I'm about to make?* Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.
3. *What could we have done to prevent this bug in the first place?* This question is the first step toward establishing a statistical software quality assurance approach (Chapter 26). If we correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

## 13.8 Summary

Software testing accounts for the largest percentage of technical effort in the software process. Yet we are only beginning to understand the subtleties of systematic test planning, execution, and control.

The objective of software testing is to uncover errors. To fulfill this objective, a series of test steps—unit, integration, validation, and system tests—are planned and executed. Unit and integration tests concentrate on functional verification of a component and incorporation of components into the software architecture. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system.

Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened.

Unlike testing (a systematic, planned activity), debugging must be viewed as an art. Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error. Of the many resources available during debugging, the most valuable is the counsel of other members of the software engineering staff.

The requirement for higher-quality software demands a more systematic approach to testing. To quote Dunn and Ullman [DUN82], What is required is an overall strategy, spanning the strategic test space, quite as deliberate in its methodology as was the systematic development on which analysis, design and code were based.

In this chapter, we have examined the strategic test space, considering the steps that have the highest likelihood of meeting the overriding test objective: to find and remove errors in an orderly and effective manner.

## REFERENCES

[BAE97] Baecker, R., C. DiGiano, and A. Marcus, "Software Visualization for Debugging," *Communications of the ACM,* vol. 40 ,no. 4, April 1997, pp. 44–54, and other papers in the same issue.

[BEI84] Beizer, B., *Software System Testing and Quality Assurance,* Van Nostrand-Reinhold, 1984.

[BER93] Berard, E., *Essays on Object-Oriented Software Engineering,* vol. 1, Addison-Wesley, 1993.

[BIN94] Binder, R., "Testing Object-Oriented Systems: A Status Report," *American Programmer,* vol. 7, no. 4, April 1994, pp. 23–28.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981, p. 37.

[BRA85] Bradley, J. H., "The Science and Art of Debugging," *Computerworld,* August 19, 1985, pp. 35–38.

[CHE90] Cheung, W. H., J. P. Black, and E. Manning, "A Framework for Distributed Debugging," *IEEE Software,* January 1990, pp. 106–115.

[DUN82] Dunn, R., and R. Ullman, *Quality Assurance for Computer Software,* McGraw-Hill, 1982, p. 158.

[GIL95] Gilb, T., "What We Fail to Do in Our Current Testing Culture," *Testing Techniques Newsletter,* (on-line edition, ttn@soft.com), Software Research, January 1995.

[HAI02] Hailpern, B., and P. Santhanam, "Software Debugging, Testing and Verification," IBM Systems Journal, vol. 41, no. 1, 2002, available at http://www.research.ibm.com/journal/sj/411/hailpern.html

[IEE01] *Software Reliability Engineering, 12th International Symposium,* IEEE, 2001.

[MCO96] McConnell, S., "Best Practices: Daily Build and Smoke Test," *IEEE Software,* vol. 13, no. 4, July 1996, pp. 143–144.

[MIL77] Miller, E., "The Philosophy of Testing," in *Program Testing Techniques,* IEEE Computer Society Press, 1977, pp. 1-3.

[MUS89] Musa, J. D., and A. F. Ackerman, "Quantifying Software Validation: When to Stop Testing?" *IEEE Software,* May 1989, pp. 19–27.

[MYE79] Myers, G., *The Art of Software Testing,* Wiley, 1979.

[SHO83] Shooman, M. L., *Software Engineering,* McGraw-Hill, 1983.

[SHN80] Shneiderman, B., *Software Psychology,* Winthrop Publishers, 1980, p. 28.

[SIN99] Singpurwalla, N., and S. Wilson, *Statistical Methods in Software Engineering: Reliability and Risk,* Springer-Verlag, 1999.

[VAN89] Van Vleck, T., "Three Questions About Each Bug You Find," *ACM Software Engineering Notes,* vol. 14, no. 5, July 1989, pp. 62–63.

[WAL89] Wallace, D. R., and R. U. Fujii, "Software Verification and Validation: An Overview," *IEEE Software,* May 1989, pp. 10–17.

[YOU75] Yourdon, E., *Techniques of Program Structure and Design,* Prentice-Hall, 1975.

## PROBLEMS AND POINTS TO PONDER

**13.1.** List some problems that might be associated with the creation of an independent test group. Are an ITG and an SQA group made up of the same people?

**13.2.** Using your own words, describe the difference between verification and validation. Do both make use of test case design methods and testing strategies?

**13.3.** Why is a highly coupled module difficult to unit test?

**13.4.** Who should perform the validation test—the software developer or the software user? Justify your answer.

**13.5.** Is it always possible to develop a strategy for testing software that uses the sequence of testing steps described in Section 13.1.3? What possible complications might arise for embedded systems?

**13.6.** As a class project, develop a *Debugging Guide* for your installation. The guide should provide language and system-oriented hints that have been learned through the school of hard knocks! Begin with an outline of topics that will be reviewed by the class and your instructor. Publish the guide for others in your local environment.

**13.7.** How can project scheduling affect integration testing?

**13.8.** The concept of "antibugging" (Section 13.3.1) is an extremely effective way to provide built-in debugging assistance when an error is uncovered:

a. Develop a set of guidelines for antibugging.
b. Discuss advantages of using the technique.
c. Discuss disadvantages of using the technique.

**13.9.** Develop a complete test strategy for the *SafeHome* system discussed throughout this book. Document it in a *Test Specification.*

**13.10.** Is unit testing possible or even desirable in all circumstances? Provide examples to justify your answer.

## Further Readings and Information Sources

Virtually every book on software testing discusses strategies along with methods for test case design. Craig and Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), Whittaker (*How to Break Software,* Addison-Wesley, 2002), Jorgensen (*Software Testing: A Craftman's Approach,* CRC Press, 2002), Splaine and his colleagues (*The Web Testing Handbook,* Software Quality Engineering Publishing, 2001), Patton (*Software Testing,* Sams Publishing, 2000), Kaner and his colleagues (*Testing Computer Software,* second edition, Wiley, 1999) all discuss testing principles, concepts, strategies and methods. Books by Black (*Managing the Testing Process,* Microsoft Press, 1999) and Perry (*Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach,* Dorset House, 1997) also address software testing strategies.

For those readers with interest in agile software development methods, Crispin and House (*Testing Extreme Programming,* Addison-Wesley, 2002) and Beck (*Test Driven Development: By Example,* Addison-Wesley, 2002) present testing strategies and tactics for Extreme Programming. Kamer and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001) present a collection of over 300 pragmatic "lessons" (guidelines) that every software tester should learn. Watkins (*Testing IT: An Off-the Shelf Testing Process,* Cambridge University Press, 2001) establishes an effective testing framework for all types of developed and acquired software.

Lewis (*Software Testing and Continuous Quality Improvement,* CRC Press, 2000) and Koomen and his colleagues (*Test Process Improvement,* Addison-Wesley, 1999) discuss strategies for continuously improving the testing process.

Sykes and McGregor (*Practical Guide to Testing Object-Oriented Software,* Addison-Wesley, 2001), Bashir and Goel (*Testing Object-Oriented Software,* Springer-Verlag, 2000), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999), Kung and his colleagues (*Testing Object-Oriented Software,* IEEE Computer Society Press, 1998), and Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) present strategies and methods for testing OO systems.

Guidelines for debugging are contained in a books by Agans (*Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Hardware and Software Problems,* AMACON, 2002), Tells and Hsieh (*The Science of Debugging,* The Coreolis Group, 2001), Robbins (*Debugging Applications,* Microsoft Press, 2000), and Dunn (*Software Defect Removal,* McGraw-Hill, 1984). Rosenberg (*How Debuggers Work,* Wiley, 1996) addresses the technology for debugging tools. Younessi (*Object-Oriented Defect Management of Software,* Prentice-Hall, 2002) presents techniques for managing defects that are encountered in object-oriented systems. Beizer [BEI84] presents an interesting "taxonomy of bugs" that can lead to effective methods for test planning. Ball (*Debugging Embedded Microprocessor Systems,* Newnes Publishing, 1998) addresses the special nature of debugging for embedded microprocessor software.

A wide variety of information sources on software testing strategies are available on the Internet. An up-to-date list of World Wide Web references that are relevant to software testing strategies can be found at the SEPA Web site:
**http://www.mhhe.com/pressman.**

CHAPTER

# 14 TESTING TACTICS

**KEY CONCEPTS**

- **BVA**
- **cyclomatic complexity**
- **equivalence partitioning**
- **flow graphs**
- **patterns**
- **testability**
- **testing**
  - **basis path**
  - **black-box**
  - **class-level**
  - **control-structure**
  - **fault-based**
  - **loops**
  - **object-oriented**
  - **scenario-based**
  - **white-box**

Testing presents an interesting anomaly for the software engineers, who by their nature are constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and then work hard to design test cases to "break" the software. Beizer [BEI90] describes this situation effectively when he states:

There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top-down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For failing to achieve inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For failing to be telepathic? For not solving human communications problems that have been kicked around . . . for forty centuries?

Should testing instill guilt? Is testing really destructive? The answer to these questions is No!

In this chapter, we discuss techniques for software test case design. Test case design focuses on a set of techniques for the creation of test cases that meet overall testing objectives and the testing strategies discussed in Chapter 13.

**QUICK LOOK**

**What is it?** Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic and interfaces of every software component, and (2) exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.

**Who does it?** During early stages of testing, a software engineer performs all tests. However, as the testing progresses, testing specialists may become involved.

**Why is it important?** Reviews and other SQA activities can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

**What are the steps?** For conventional applications, software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test case design techniques. Software requirements are exercised using "black box" test case design techniques. For object-oriented applications, "testing" begins prior to the existence of source code, but once code has been generated, a series of tests are designed to exercise operations with a class and examine whether errors exist as one class collaborates with others. As classes are integrated to form a subsystem, use-based testing, along with fault-based approaches, is applied to fully exercise collaborating classes. Finally, use-cases assist in the design of tests to uncover errors at the software validation level. In every case, the intent is to find the maximum number of errors with the minimum amount of effort and time.

**What is the work product?** A set of test cases designed to exercise both internal logic, interfaces, component collaborations, and external requirements is designed and documented, expected results are defined, and actual results are recorded.

**How do I ensure that I've done it right?** When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness. In addition, you can evaluate test coverage and track error detection activities.

## 14.1 Software Testing Fundamentals

Fundamental testing goals and principles were discussed in Chapter 5. Recall that the goal of testing is to find errors and that a good test is one that has a high probability of finding an error. Therefore, a software engineer should design and implement a computer-based system or a product with "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

"Every program does something right; it just may not be the thing we want it to do."

**Author unknown**

**Testability.** James Bach[1] provides the following definition for testability: "*Software testability* is simply how easily [a computer program] can be tested." The following characteristics lead to testable software.

**What are the characteristics of testability?**

**Operability.** "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

1 The paragraphs that follow are used with permission of James Bach (copyright 1994) and have been adapted from material that originally appeared in a posting in the newsgroup comp.software-eng.

**Observability.** "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

**Controllability.** "The better we can control the software, the more the testing can be automated and optimized." Software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.

**Decomposability.** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." The software system is built from independent modules that can be tested independently.

**Simplicity.** "The less there is to test, the more quickly we can test it." The program should exhibit *functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements), *structural simplicity* (e.g., architecture is modularized to limit the propagation of faults), and *code simplicity* (e.g., a coding standard is adopted for ease of inspection and maintenance).

**Stability.** "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.

**Understandability.** "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

The attributes suggested by Bach can be used by a software engineer to develop a software configuration (i.e., programs, data, and documents) that is amenable to testing.

**"Errors are more common, more pervasive, and more troublesome in software than with other technologies."**
**David Parnas**

**Test characteristics.** And what about the tests themselves? Kaner, Falk, and Nguyen [KAN93] suggest the following attributes of a "good" test:

**What is a "good" test?**

1. *A good test has a high probability of finding an error.* To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

2. *A good test is not redundant.* Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).
3. *A good test should be "best of breed"* [KAN93]. In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
4. *A good test should be neither too simple nor too complex.* Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

SAFEHOME

***Designing Unique Tests***

**The scene:** Vinod's cubical.

**The players:** Vinod and Ed—members of the SafeHome software engineering team.

**The conversation:**

**Vinod:** So these are the test cases you intend to run for the *passwordValidation* operation.

**Ed:** Yeah, they should cover pretty much all possibilities for the kinds of passwords a user might enter.

**Vinod:** So let's see . . . you note that the correct password will be 8080, right?

**Ed:** Uh huh.

**Vinod:** And you specify passwords 1234 and 6789 to test for error in recognizing invalid passwords?

**Ed:** Right, and I also test passwords that are close to the correct password, see . . . 8081 and 8180.

**Vinod:** Those are okay, but I don't see much point in running both the 1234 and 6789 inputs. They're redundant . . . test the same thing, don't they?

**Ed:** Well, they're different values.

**Vinod:** That's true, but if 1234 doesn't uncover an error . . . in other words . . . the *passwordValidation* operation notes that it's an invalid password, it is not likely that 6789 will show us anything new.

**Ed:** I see what you mean.

**Vinod:** I'm not trying to be picky here . . . it's just that we have limited time to do testing, so it's a good idea to run tests that have a high likelihood of finding new errors.

**Ed:** Not a problem . . . I'll give this a bit more thought.

## 14.2 Black-Box and White-Box Testing

Any engineered product (and most other things) can be tested in one of two ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function; (2) knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh"; that is, internal operations are performed according to specifications, and all internal components have been adequately exercised. The first test approach is called black-box testing and the second, white-box testing.[2]

*Black-box testing* alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software. *White-box testing* of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by providing test cases that exercise specific sets of conditions and/or loops.

**"There is only one rule in designing test cases: cover all features, but do not make too many test cases."**

**Tsuneo Yamaura**

**KEY POINT**

White-box tests can be designed only after component-level design (or source code) exists. The logical details of the program must be available.

At first glance it would seem that very thorough white-box testing would lead to 100 percent correct programs. All we need to do is identify all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems (see the sidebar discussion). White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

INFO

***Exhaustive Testing***

Consider the 100-line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately $10^{14}$ possible paths that may be executed in this program!

To put this number into perspective, we assume that a magic test processor ("magic" because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules.

Therefore, it is reasonable to assert that exhaustive testing is impossible for large software systems.

## 14.3 White-Box Testing

*White-box testing,* sometimes called *glass-box testing,* is a test case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a module have been ex-

2 The terms *functional testing* and *structural testing* are sometimes used in place of black-box and white-box testing, respectively.

ercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

"Bugs lurk in corners and congregate at boundaries."

**Boris Beizer**

## 14.4 Basis Path Testing

*Basis path testing* is a white-box testing technique first proposed by Tom McCabe [MCC76]. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

### 14.4.1 Flow Graph Notation

ADVICE

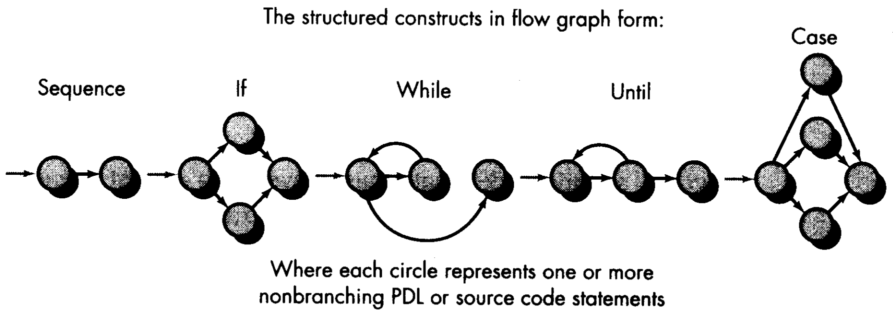
*A flow graph should be drawn only when the logical structure of a component is complex. The flow graph allows you to trace program paths more readily.*

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced.[3] The flow graph depicts logical control flow using the notation illustrated in Figure 14.1. Each structured construct (Chapter 11) has a corresponding flow graph symbol.

To illustrate the use of a flow graph, we consider the procedural design representation in Figure 14.2a. Here, a flowchart is used to depict program control structure. Figure 14.2b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 14.2b, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links,* represent

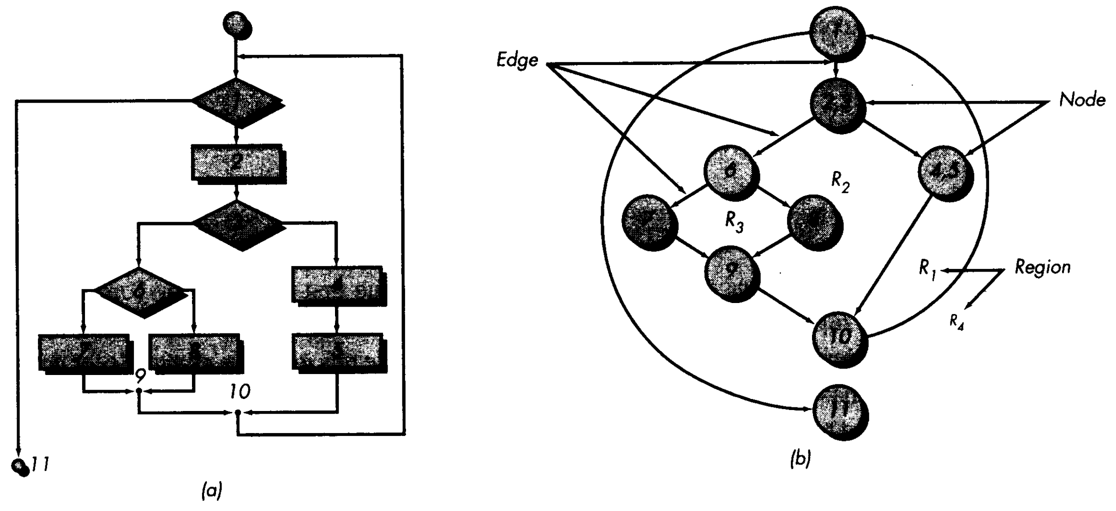
FIGURE 14.1

**Flow graph notation**


The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

3 In actuality, the basis path method can be conducted without the use of flow graphs. However, they serve as a useful notation for understanding control flow and illustrating the approach.

FIGURE 14.2 (a) Flowchart and (b) flow graph



flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct in Figure 14.1). Areas bounded by edges and nodes are called *regions*. When counting regions, we include the area outside the graph as a region.[4]

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 14.3, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

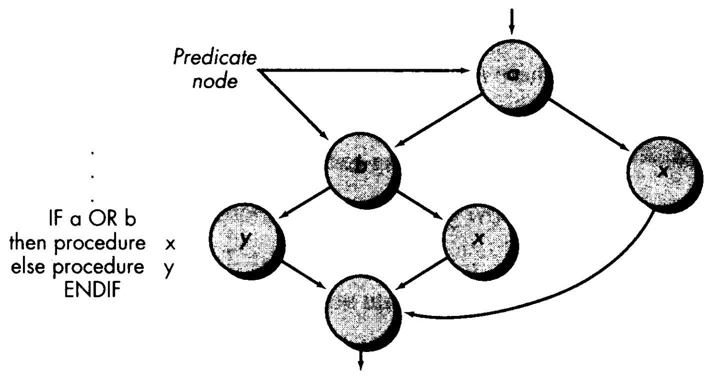
### 14.4.2 Independent Program Paths

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 14.2b is:

| | |
|---|---|
| path 1: | 1-11 |
| path 2: | 1-2-3-4-5-10-1-11 |

4 A more detailed discussion of graphs and their uses is presented in Section 14.6.1.

**FIGURE 14.3**
**Compound logic**



path 3: 1-2-3-6-8-9-10-1-11

path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

ADVICE

*Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone. Use it for test planning as well as test case design.*

Paths 1, 2, 3, and 4 constitute a *basis set* for the flow graph in Figure 14.2b. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time, and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer. *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic complexity has a foundation in graph theory and is computed in one of three ways:

**How do I compute cyclomatic complexity?**

1. The number of regions corresponds to the cyclomatic complexity.
2. Cyclomatic complexity, $V(G)$, for a flow graph, $G$, is defined as

   $V(G) = E - N + 2$

   where $E$ is the number of flow graph edges, and $N$ is the number of flow graph nodes.

KEY POINT

Cyclomatic complexity provides the upper bound on the number of test cases that will be required to guarantee that every statement in the program has been executed at least one time.

3. Cyclomatic complexity, $V(G)$, for a flow graph, $G$, is also defined as

   $V(G) = P + 1$

   where $P$ is the number of predicate nodes contained in the flow graph $G$.

Referring once more to the flow graph in Figure 14.2b, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. $V(G) = 11$ edges $- 9$ nodes $+ 2 = 4$.
3. $V(G) = 3$ predicate nodes $+ 1 = 4$.

More important, the value for $V(G)$ provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

SAFEHOME

***Using Cyclomatic Complexity***

**The scene:** Shakira's cubicle.

**The players:** Vinod and Shakira—members of the SafeHome software engineering team who are working on test planning for the security function.

**The conversation:**

**Shakira:** Look . . . I know that we should unit test all the components for the security function, but there are a lot of 'em and if you consider the number of operations that have to be exercised, I don't know . . . maybe we should forget white-box testing, integrate everything, and start running black-box tests.

**Vinod:** You figure we don't have enough time to do component tests, exercise the operations, and then integrate?

**Shakira:** The deadline for the first increment is getting closer than I'd like . . . yeah, I'm concerned.

**Vinod:** Why don't you at least run white-box tests on the operations that are likely to be the most error prone?

**Shakira (exasperated):** And exactly how do I know which are likely to be the most error prone?

**Vinod:** *V* of *G*.

**Shakira:** Huh?

**Vinod:** Cyclomatic complexity—*V* of *G*. Just compute *V*(*G*) for each of the operations within each of the components and see which have the highest values for *V*(*G*). They're the ones that are most likely to be error prone.

**Shakira:** And how do I compute *V* of *G*?

**Vinod:** It's really easy. Here's a book that describes how to do it.

**Shakira (leafing through the pages):** Okay, it doesn't look hard. I'll give it a try. The ops with the highest *V*(*G*) will be the candidates for white-box tests.

**Vinod:** Just remember that there are no guarantees. A component with a low *V*(*G*) can still be error prone.

**Shakira:** Alright. But at least this'll help me to narrow down the number of components that have to undergo white-box testing.

### 14.4.3 Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure *average*, depicted in PDL in Figure 14.4, will be used as an example to illustrate

FIGURE 14.4

**PDL with nodes identified**

```
PROCEDURE average;

  * This procedure computes the average of 100 or fewer
    numbers that lie between bounding values; it also computes the
    sum and the total number valid.

  INTERFACE RETURNS average, total.input, total.valid;
  INTERFACE ACCEPTS value, minimum, maximum;

  TYPE value[1:100] IS SCALAR ARRAY;
  TYPE average, total.input, total.valid;
     minimum, maximum, sum IS SCALAR;
  TYPE i IS INTEGER;
   { i = 1;
 1 { total.input = total.valid = 0;    2
   { sum = 0;                        ↙
   { DO WHILE value[i] <> -999 AND total.input < 100   3
    4 increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
  5 →     { THEN increment total.valid by 1;
        7 {        sum = s sum + value[i]
          { ELSE skip
   8 { ENDIF
     { increment i by 1;
  9 ENDDO
    IF total.valid > 0   10
    11 THEN average = sum / total.valid;
 12 → ELSE average = -999;
  13 ENDIF
END average
```

each step in the test case design method. Note that *average,* although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

1. **Using the design or code as a foundation, draw a corresponding flow graph.** A flow graph is created using the symbols and construction rules presented in Section 14.4.1. Referring to the PDL for *average* in Figure 14.4, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 14.5.
2. **Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity, $V(G)$, is determined by applying the algorithms described in Section 14.4.2. It should be noted that $V(G)$ can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1. Referring to Figure 14.5,

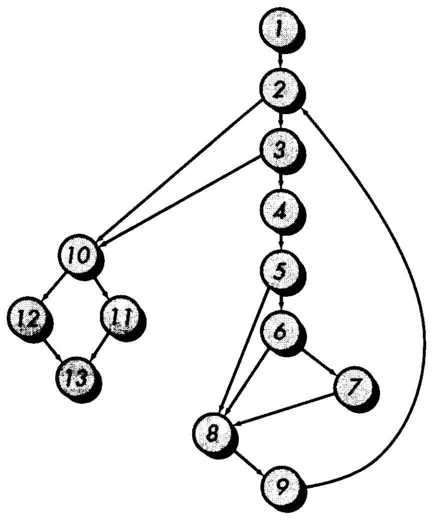
   $V(G) = 6$ regions
   $V(G) = 17$ edges $- 13$ nodes $+ 2 = 6$
   $V(G) = 5$ predicate nodes $+ 1 = 6$

"To err is human, to find a bug is divine."

Robert Dunn

FIGURE 14.5

**Flow graph for the procedure *average***



3. **Determine a basis set of linearly independent paths.** The value of $V(G)$ provides the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify six paths:

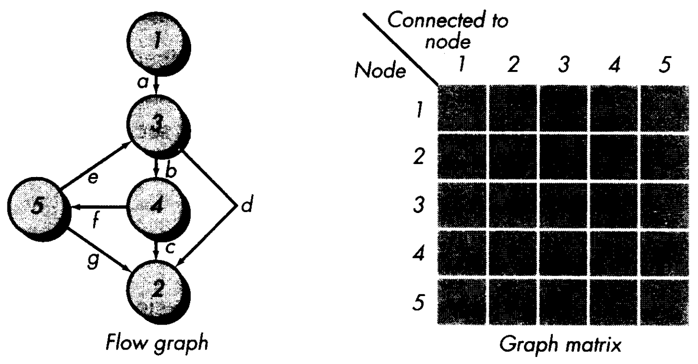
| | |
|---|---|
| path 1: | 1-2-10-11-13 |
| path 2: | 1-2-10-12-13 |
| path 3: | 1-2-3-10-11-13 |
| path 4: | 1-2-3-4-5-8-9-2-. . . |
| path 5: | 1-2-3-4-5-6-8-9-2-. . . |
| path 6: | 1-2-3-4-5-6-7-8-9-2-. . . |

   The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. **Prepare test cases that will force execution of each path in the basis set.** Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

**FIGURE 14.6**
**Graph matrix**



## 14.4.4 Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a *graph matrix,* can be quite useful.

A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix [BEI90] is shown in Figure 14.6.

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.

**What is a graph matrix, and how do we extend it for use in testing?**

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link.
- The memory required during traversal of a link.
- The resources required during traversal of a link.

Beizer [BEI90] provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

"Paying more attention to running tests than to designing them is a classic mistake."
**Brian Marick**

## 14.5 Control Structure Testing

The basis path testing technique described in Section 14.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and effective, it is not sufficient in itself. In this section, variations on control structure testing are discussed briefly. These broaden testing coverage and improve quality of white-box testing.

### 14.5.1 Condition Testing

KEY POINT
Errors are much more common in the neighborhood of logical conditions than they are in the locus of sequential processing statements.

*Condition testing* [TAI89] is a test case design method that exercises the logical conditions contained in a program module. *A simple condition* is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

$E_1$ <relational-operator> $E_2$

where $E_1$ and $E_2$ are arithmetic expressions and <relational-operator> is one of the following: $<$, $\leq$, $=$, $\neq$ (nonequality), $>$, or $\geq$. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬). A condition without relational expressions is referred to as a Boolean expression. Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of parentheses (surrounding a simple or compound Boolean condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

### 14.5.2 Data Flow Testing

The *data flow testing* method selects test paths of a program according to the locations of definitions and uses of variables in the program.To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with $S$ as its statement number,

DEF($S$) = $\{X \mid$ statement $S$ contains a definition of $X\}$
USE($S$) = $\{X \mid$ statement $S$ contains a use of $X\}$

If statement $S$ is an *if* or *loop* statement, its DEF set is empty and its USE set is based on the condition of statement $S$. The definition of variable $X$ at statement $S$ is said to be live at statement $S'$ if there exists a path from statement $S$ to statement $S'$ that contains no other definition of $X$.

A *definition-use (DU) chain* of variable $X$ is of the form $[X, S, S']$, where $S$ and $S'$ are statement numbers, $X$ is in DEF($S$) and USE($S'$), and the definition of $X$ in statement $S$ is live at statement $S'$.

ADVICE

*It is unrealistic to assume that data flow testing will be used extensively when testing a large system. However, it can be used in a targeted fashion for areas of software that are suspect.*

One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the *DU testing strategy*. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the *then part* has no definition of any variable and the *else part* does not exist. In this situation, the else branch of the *if* statement is not necessarily covered by DU testing. A number of data flow testing strategies have been studied and compared (e.g., [FRA88], [NTA88], [FRA93]). The interested reader is urged to consider these other references.

**"Good testers are masters at noticing 'something funny' and acting on it."**

**Brian Marick**

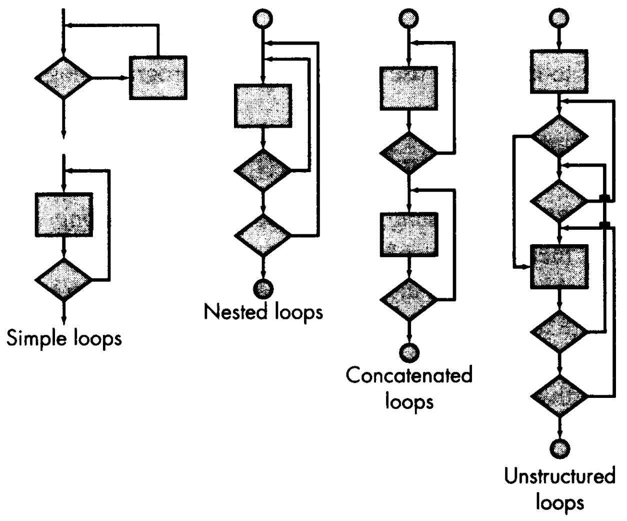
## 14.5.3 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

*Loop testing* is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [BEI90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 14.7).

**FIGURE 14.7**

**Classes of loops**



**Simple loops.** The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. *m* passes through the loop where $m < n$.
5. $n - 1, n, n + 1$ passes through the loop.

**Nested loops.** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increased. This would result in an impractical number of tests. Beizer [BEI90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

**Concatenated loops.** Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

ADVICE

*You can't test unstructured loops effectively. Redesign them.*

**Unstructured loops.** Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 11).

## 14.6 Black-Box Testing

*Black-box testing*, also called *behavioral testing*, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing (see Chapter 13). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

**What questions do black-box tests answer?**

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria [MYE79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

## 14.6.1 Graph-Based Testing Methods

**KEY POINT**

A graph represents the relationships between data objects and program objects, enabling us to derive test cases that search for errors associated with these relationships.

The first step in black-box testing is to understand the objects[5] that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another" [BEI95]. Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

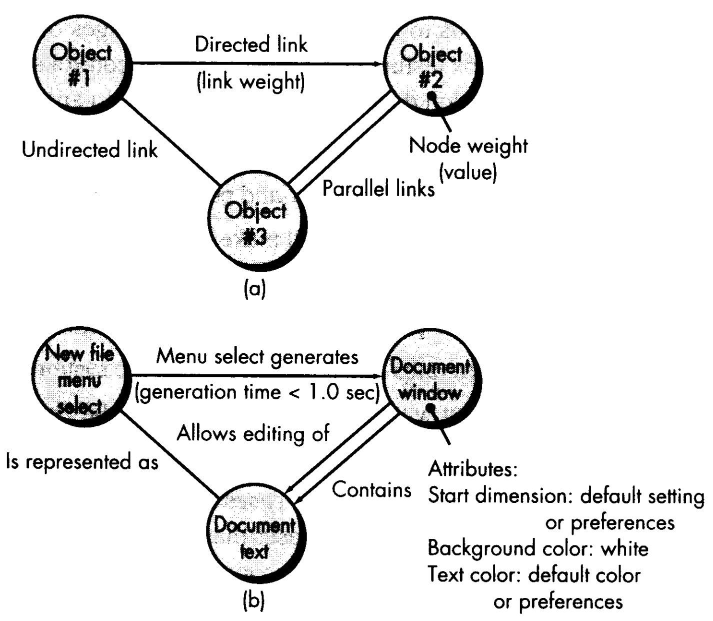
To accomplish these steps, the software engineer begins by creating a *graph*—a collection of *nodes* that represent objects; *links* that represent the relationships between objects; *node weights* that describe the properties of a node (e.g., a specific data value or state behavior); and *link weights* that describe some characteristic of a link.

The symbolic representation of a graph is shown in Figure 14.8a. Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

5 In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

FIGURE 14.8

**(a) Graph notation, (b) simple example**



As a simple example, consider a portion of a graph for a word-processing application (Figure 14.8b) where

*Object #1* = **newFile** (menu selection)

*Object #2* = **documentWindow**

*Object #3* = **documentText**

Referring to the figure, a menu select on **newFile** generates a document window. The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.** In reality, a far more detailed graph would have to be generated as a precursor to test case design. The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

Beizer [BEI95] describes a number of behavioral testing methods that can make use of graphs:

**Transaction flow modeling.** The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps. The data flow diagram (Chapter 8) can be used to assist in creating graphs of this type.

**Finite state modeling.** The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state. The state diagram (Chapter 8) can be used to assist in creating graphs of this type.

**Data flow modeling.** The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node **FICA tax withheld** (**FTW**) is computed from **gross wages** (**GW**) using the relationship, **FTW = 0.62 × GW.**

**Timing modeling.** The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. The interested reader should see [BEI95] for comprehensive coverage.

### 14.6.2 Equivalence Partitioning

*Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

ADVICE

*Input classes are known relatively early in the software process. For this reason, begin thinking about equivalence partitioning as the design is created.*

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [BEI95]. An *equivalence class* represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

**How do I define equivalence classes for testing?**

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying these guidelines for the derivation of equivalence classes, test cases for each input domain data object can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

### 14.6.3 Boundary Value Analysis

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. BVA leads to a selection of test cases that exercise bounding values.

**KEY POINT**
BVA extends equivalence partitioning by focusing on data at the "edges" of an equivalence class.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [MYE79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

**How do I create BVA test cases?**

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

**"The Ariane 5 rocket blew up on lift-off due solely to a software defect (a bug) involving the conversion of a 64-bit floating point value into a 16-bit integer. The rocket and its four satellites were *uninsured* and worth $500 million. A comprehensive system test would have found the bug but was vetoed for budgetary reasons."**

**A news report**

### 14.6.4 Orthogonal Array Testing

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test processing of the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

*Orthogonal array testing* can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding errors associated with *region faults*—an error category associated with faulty logic within a software component.

**KEY POINT**

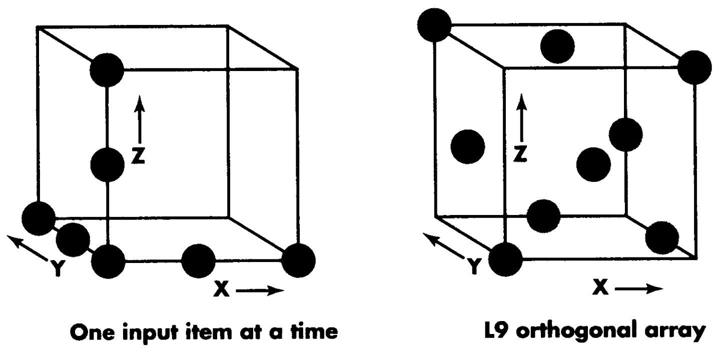
Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases.

To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, *X, Y,* and *Z*. Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases. Phadke [PHA97] suggests a geometric view of the possible test cases associated with *X, Y,* and *Z* illustrated in Figure 14.9. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

When orthogonal array testing occurs, an *L9 orthogonal array* of test cases is created. The L9 orthogonal array has a "balancing property [PHA97]." That is, test cases (represented by blue dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure 14.9. Test coverage across the input domain is more complete.

**FIGURE 14.9**

**A geometric view of test cases [PHA97]**



To illustrate the use of the L9 orthogonal array, consider the *send* function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now
P1 = 2, send it one hour later
P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). Phadke [PHA97] assesses these test cases by stating:

Such test cases are useful only when one is certain that these test parameters do not interact. They can detect logic faults where a single parameter value makes the software malfunction. These faults are called *single mode faults.* This method cannot detect logic faults that cause malfunction when two or more parameters simultaneously take certain values; that is, it cannot detect any interactions. Thus its ability to detect faults is limited.

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is $3^4 = 81$, large, but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables us to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax *send* function is illustrated in Figure 14.10.

**FIGURE 14.10**

**An L9 orthogonal array**

| Test case | Test parameters | | | |
|---|---|---|---|---|
| | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 |
| 3 | 1 | 3 | 3 | 3 |
| 4 | 2 | 1 | 2 | 3 |
| 5 | 2 | 2 | 3 | 1 |
| 6 | 2 | 3 | 1 | 2 |
| 7 | 3 | 1 | 3 | 2 |
| 8 | 3 | 2 | 1 | 3 |
| 9 | 3 | 3 | 2 | 1 |

Phadke [PHA97] assesses the result of tests using the L9 orthogonal array in the following manner:

**Detect and isolate all single mode faults.** A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example, tests 1, 2, and 3 [Figure 14.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.

**Detect all double mode faults.** If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault.* Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.

**Multimode faults.** Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multimode faults are also detected by these tests.

A detailed discussion of orthogonal array testing can be found in [PHA89].

SOFTWARE TOOLS

### *Test Case Design*

**Objective:** To assist the software team in developing a complete set of test cases for both black-box and white-box testing.

**Mechanics:** These tools fall into two broad categories: static testing and dynamic testing. Three different types of static testing tools are used in the industry: code-based testing tools, specialized testing languages, and requirements-based testing tools. Code-based testing tools accept source code as input and perform a number of analyses that result in the generation of test cases. Specialized testing languages (e.g., ATLAS) enable a software engineer to write detailed test specifications that describe each test case and the logistics for its execution. Requirements-based testing tools isolate specific user requirements and suggest test cases (or classes of tests) that will exercise the requirements. Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise instrumenting the execution flow of the program.

**Representative Tools**[6]

*McCabe Test,* developed by McCabe & Associates (www.mccabe.com), implements a variety of path testing techniques derived from an assessment of cyclomatic complexity and other software metrics.

*Panorama,* developed by International Software Automation, Inc. (www.softwareautomation.com), encompasses a complete set of tools for object-oriented software development including tools that assist test case design and test planning.

*TestWorks,* developed by Software Research, Inc. (www.soft.com/Products), is a complete set of automated testing tools that assists in the design of test cases for software developed in C/C++ and Java and provides support for regression testing.

*T-Vec Test Generation System,* developed by T-VEC Technologies (www.t-vec.com), is a tool set that supports unit, integration, and validation testing by assisting in the design of test cases using information contained in an OO requirements specification.

6 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

## 14.7 Object-Oriented Testing Methods

The architecture of object-oriented software results in a series of layered subsystems that encapsulate collaborating classes. Each of these system elements (subsystems and classes) perform functions that help to achieve system requirements. It is necessary to test an OO system at a variety of different levels to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers.

Object-oriented testing is strategically similar to the testing of conventional systems, but it is tactically different. Because OO analysis and design models are similar in structure and content to the resultant OO program, "testing" can begin with the review of these models. Once code has been generated, actual OO testing begins "in the small" with a series of tests designed to exercise class operations and examine whether errors exist as one class collaborates with other classes. As classes are integrated to form a subsystem, use-based testing, along with fault-based approaches, is applied to fully exercise collaborating classes. Finally, use-cases are used to uncover errors at the software validation level.

Conventional test case design is driven by an input-process-output view of software or the algorithmic detail of individual modules. Object-oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class.

### 14.7.1 The Test Case Design Implications of OO Concepts

**WebRef**
An excellent collection of papers and resources on OO testing can be found at **www.rbsc.com.**

As a class evolves through the analysis and design models, it becomes a target for test case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing. As Binder [BIN94] notes, "Testing requires reporting on the concrete and abstract state of an object." Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

Inheritance also leads to additional challenges for the test case designer. We have already noted that each new context of usage requires retesting, even though reuse has been achieved. In addition, multiple inheritance[7] complicates testing further by increasing the number of contexts for which testing is required [BIN94]. If subclasses instantiated from a superclass are used within the same problem domain, it is likely that the set of test cases derived for the superclass can be used when testing the subclass. However, if the subclass is used in an entirely different context, the superclass test cases will have little applicability and a new set of tests must be designed.

7 An OO concept that should be used with extreme care.

### 14.7.2 Applicability of Conventional Test Case Design Methods

The white-box testing methods described in earlier sections can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested. However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods. As we noted earlier in this chapter, use-cases can provide useful input in the design of black-box and state-based tests [AMB95].

### 14.7.3 Fault-Based Testing[8]

**KEY POINT**
The strategy for fault-based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis.

The objective of *fault-based testing* within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, the preliminary planning required to perform fault-based testing begins with the analysis model. The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

Of course, the effectiveness of these techniques depends on how testers perceive a plausible fault. If real faults in an OO system are perceived to be implausible, then this approach is really no better than any random testing technique. However, if the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find significant numbers of errors with relatively low expenditures of effort.

**What types of faults are encountered in operation calls and message connections?**

Integration testing (when applied in an OO context) looks for plausible faults in operation calls or message connections. Three types of faults are encountered in this context: unexpected result, wrong operation/message used, incorrect invocation. To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined.

Integration testing applies to attributes as well as to operations. The "behaviors" of an object are defined by the values that its attributes are assigned. Testing should exercise the attributes to determine whether proper values occur for distinct types of object behavior.

It is important to note that integration testing attempts to find errors in the client object, not the server. Stated in conventional terms, the focus of integration testing is

8 Sections 14.7.3 through 14.7.6 have been adapted from an article by Brian Marick posted on the Internet newsgroup **comp.testing.** This adaptation is included with the permission of the author. For further information on these topics, see [MAR94]. It should be noted that the techniques discussed in Sections 14.7.3 through 14.7.6 are also applicable for conventional software.

to determine whether errors exist in the calling code, not the called code. The operation call is used as a clue, a way to find test requirements that exercise the calling code.

"If you want and expect a program to work, you will more likely see a working program—you will miss failures."
**Cem Kaner et al.**

### 14.7.4 Test Cases and Class Hierarchy

Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process. Consider the following situation. A class **Base** contains operations *inherited()* and *redefined()*. A class **Derived** redefines *redefined()* to serve in a local context. There is little doubt the **Derived::redefined()** has to be tested because it represents a new design and new code. But does **Derived::inherited()** have to be retested?

**KEY POINT**
Even though a base class has been thoroughly tested, you will still have to test all classes derived from it.

If **Derived::inherited()** calls *redefined()* and the behavior of *redefined()* has changed, **Derived::inherited()** may mishandle the new behavior. Therefore, it needs new tests even though the design and code have not changed. It is important to note, however, that only a subset of all tests for **Derived::inherited()** may have to be conducted. If part of the design and code for *inherited()* does not depend on *redefined()* (i.e., that does not call it, nor any code that indirectly calls it), that code need not be retested in the derived class.

**Base::redefined()** and **Derived::redefined()** are two different operations with different specifications and implementations. Each would have a set of test requirements derived from the specification and implementation. Those test requirements probe for plausible faults: integration faults, condition faults, boundary faults, and so forth. But the operations are likely to be similar. Their sets of test requirements will overlap. The better the OO design, the greater is the overlap. New tests need to be derived only for those **Derived::redefined()** requirements that are not satisfied by the **Base::redefined()** tests.

To summarize, the **Base::redefined()** tests are applied to objects of class **Derived.** Test inputs may be appropriate for both base and derived classes, but the expected results may differ in the derived class.

### 14.7.5 Scenario-Based Testing

Fault-based testing misses two main types of errors: (1) incorrect specifications and (2) interactions among subsystems. When errors associated with incorrect specifications occur, the product doesn't do what the customer wants. It might do the wrong thing, or it might omit important functionality. But in either circumstance, quality (conformance to requirements) suffers. Errors associated with subsystem interactions occur when the behavior of one subsystem creates circumstances (e.g., events, data flow) that cause another subsystem to fail.

**KEY POINT**
Scenario-based testing will uncover errors that occur when any actor interacts with the software.

*Scenario-based testing* concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

Scenarios uncover interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test (users do not limit themselves to the use of one subsystem at a time).

As an example, consider the design of scenario-based tests for a text editor by reviewing the informal use-cases that follow:

**Use-Case:** *Fix the Final Draft*

**Background:** It's not unusual to print the "final" draft, read it, and discover some annoying errors that weren't obvious from the on-screen image. This use-case describes the sequence of events that occurs when this happens.

1. Print the entire document.
2. Move around in the document, changing certain pages.
3. As each page is changed, it's printed.
4. Sometimes a series of pages is printed.

This scenario describes two things: a test and specific user needs. The user needs are obvious: (1) a method for printing single pages and (2) a method for printing a range of pages. As far as testing goes, there is a need to test editing after printing (as well as the reverse). The tester hopes to discover that the printing function causes errors in the editing function; that is, that the two software functions are not properly independent.

ADVICE

*Although scenario-based testing has merit, you will get a higher return on time invested by reviewing use-cases when they are developed as part of the analysis model.*

**Use-Case:** *Print a New Copy*

**Background:** Someone asks the user for a fresh copy of the document. It must be printed.

1. Open the document.
2. Print it.
3. Close the document.

Again, the testing approach is relatively obvious, except that this document didn't appear out of nowhere. It was created in an earlier task. Does that task affect this one?

In many modern editors, documents remember how they were last printed. By default, they print the same way the next time. After the *Fix the Final Draft* scenario, just selecting "Print" in the menu and clicking the Print button in the dialog box will cause the last corrected page to print again. So, according to the editor, the correct scenario should look like this:

**Use-Case:** *Print a New Copy*

1. Open the document.
2. Select "Print" in the menu.
3. Check if you're printing a page range; if so, click to print the entire document.
4. Click on the Print button.
5. Close the document.

But this scenario indicates a potential specification error. The editor does not do what the user reasonably expects it to do. Customers will often overlook the check noted in step 3 above. They will then be annoyed when they trot off to the printer and find one page when they wanted 100. Annoyed customers signal specification bugs.

A test case designer might miss this dependency in test design, but it is likely that the problem would surface during testing. The tester would then have to contend with the probable response, "That's the way it's supposed to work!"

### 14.7.6 Testing Surface Structure and Deep Structure

*Surface structure* refers to the externally observable structure of an OO program. That is, the structure that is immediately obvious to an end-user. Rather than performing functions, the users of many OO systems may be given objects to manipulate in some way. But whatever the interface, tests are still based on user tasks. Capturing these tasks involves understanding, watching, and talking with representative users (and as many nonrepresentative users as are worth considering).

**KEY POINT**
Testing surface structure is analogous to black-box testing. Deep structure testing is similar to white-box testing.

There will surely be some difference in detail. For example, in a conventional system with a command-oriented interface, the user might use the list of all commands as a testing checklist. If no test scenarios exist to exercise a command, testing has likely overlooked some user tasks (or the interface has useless commands). In an object-based interface, the tester might use the list of all objects as a testing checklist.

The best tests are derived when the designer looks at the system in a new or unconventional way. For example, if the system or product has a command-based interface, more thorough tests will be derived if the test case designer pretends that operations are independent of objects. Ask questions like, "Might the user want to use this operation—which applies only to the **Scanner** object—while working with the printer?" Whatever the interface style, test case design that exercises the surface structure should use both objects and operations as clues leading to overlooked tasks.

*Deep structure* refers to the internal technical details of an OO program. That is, the structure that is understood by examining the design and/or code. Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the design model (Chapters 9 through 12) for OO software.

The analysis and design models are used as the basis for deep structure testing. For example, the UML collaboration diagram or the deployment model depicts collaborations between objects and subsystems that may not be externally visible. The test case designer then asks: Have we captured (as a test) some task that exercises the collaboration noted on the collaboration diagram? If not, why not?

"Be not ashamed of mistakes and thus make them crimes."
Confucius

## 14.8 Testing Methods Applicable at the Class Level

In Chapter 13, we noted that software testing begins "in the small" and slowly progresses toward testing "in the large." Testing in the small focuses on a single class and the methods that are encapsulated by the class. Random testing and partitioning are methods that can be used to exercise a class during OO testing [KIR94].

### 14.8.1 Random Testing for OO Classes

ADVICE

*The number of possible permutations for random testing can grow quite large. A strategy similar to orthogonal array testing can be used to improve testing efficiency.*

To provide brief illustrations of these methods, consider a banking application in which an **Account** class has the following operations: *open(), setup(), deposit(), withdraw(), balance(), summarize(), creditLimit(),* and *close()* [KIR94]. Each of these operations may be applied for **Account,** but certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations. The minimum behavioral life history of an instance of **Account** includes the following operations:

**open • setup • deposit • withdraw • close**

This represents the minimum test sequence for **Account.** However, a wide variety of other behaviors may occur within this sequence:

**open • setup • deposit • [deposit | withdraw | balance | summarize | creditLimit]$^n$ • withdraw • close**

A variety of different operation sequences can be generated randomly. For example:

*Test case $r_1$:* **open • setup • deposit • deposit • balance • summarize • withdraw • close**

*Test case $r_2$:* **open • setup • deposit • withdraw • deposit • balance • creditLimit • withdraw • close**

These and other random order tests are conducted to exercise different class instance life histories.

SAFEHOME

***Class Testing***

**The scene:** Shakira's cubicle.

**The players:** Jamie and Shakira—members of the *SafeHome* software engineering team who are working on test case design for the security function.

**The conversation:**

**Shakira:** I've developed some tests for the **Detector** class [Figure 11.4]—you know, the one that allows access to all of the **Sensor** objects for the security function. You familiar with it?

**Jamie (laughing):** Sure, it's the one that allowed you to add the "doggie angst" sensor.

**Shakira:** The one and only. Anyway, it has an interface with four ops: *read(), enable(), disable(),* and *test().* Before a sensor can be read, it must be enabled. Once it's sensor can be read and tested. It can be disabled at any time, except if an alarm condition is being processed. So I defined a simple test sequence that will exercise its behavioral life history.

(Shows Jamie the following sequence.)

**#1: enable•test•read•disable**

**Jamie:** That'll work, but you've got to do more testing than that!

**Shakira:** I know, I know. Here are some other sequences I've come up with.

(She shows Jamie the following sequences.)

**#2: enable•test•[read]$^n$•test•disable**

**#3: [read]$^n$**

**#4: enable•disable•[test | read]**

**Jamie:** So let me see if I understand the intent of these. #1 goes through a normal life history, sort of a conventional usage. #2 repeats the read operation *n* times, and that's a likely scenario. #3 tries to read the sensor before it's been enabled . . . that should produce an error message of some kind, right? #4 enables and disables the sensor and then tries to read it. Isn't that the same as test #3?

**Shakira:** Actually no. In #4, the sensor has been enabled. What #4 really tests is whether the disable op works as it should. A read() or test() after disable() should generate the error message. If it doesn't, then we have an error in the disable op.

**Jamie:** Cool. Just remember that the four tests have to be applied for every sensor type since all the ops may be subtly different depending on the type of sensor.

**Shakira:** Not to worry. That's the plan.

### 14.8.2 Partition Testing at the Class Level

*Partition testing* reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning (Section 14.6.2) for conventional software. Input and output are categorized and test cases are designed to exercise each category. But how are the partitioning categories derived?

**What testing options are available at the class level?**

State-based partitioning categorizes class operations based on their ability to change the state of the class. Again considering the **Account** class, state operations include *deposit()* and *withdraw()*, whereas nonstate operations include *balance()*, *summarize()*, and *creditLimit()*. Tests are designed in a way that exercises operations that change state and those that do not change state separately. Therefore,

*Test case $p_1$:* **open•setup•deposit•deposit•withdraw•withdraw•close**

*Test case $p_2$:* **open•setup•deposit•summarize•creditLimit•withdraw•close**

Test case $p_1$ changes state, while test case $p_2$ exercises operations that do not change state (other than those in the minimum test sequence).

*Attribute-based partitioning* categorizes class operations based on the attributes that they use. For the **Account** class, the attributes **balance** and **creditLimit** can be used to define partitions. Operations are divided into three partitions: (1) operations that use **creditLimit**, (2) operations that modify **creditLimit**, and (3) operations that do not use or modify **creditLimit**. Test sequences are then designed for each partition.

*Category-based partitioning* categorizes class operations based on the generic function that each performs. For example, operations in the **Account** class can be categorized as initialization operations—*open(), setup()*, computational operations—*deposit(), withdraw()*, queries—*balance(), summarize(), creditLimit()*) and termination operations—*close()*.

## 14.9 InterClass Test Case Design

Test case design becomes more complicated as integration of the object-oriented system begins. It is at this stage that testing of collaborations between classes must begin. To illustrate "interclass test case generation" [KIR94], we expand the banking example introduced in Section 14.8 to include the classes and collaborations noted in Figure 14.11. The direction of the arrows in the figure indicates the direction of messages, and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

Like the testing of individual classes, class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.

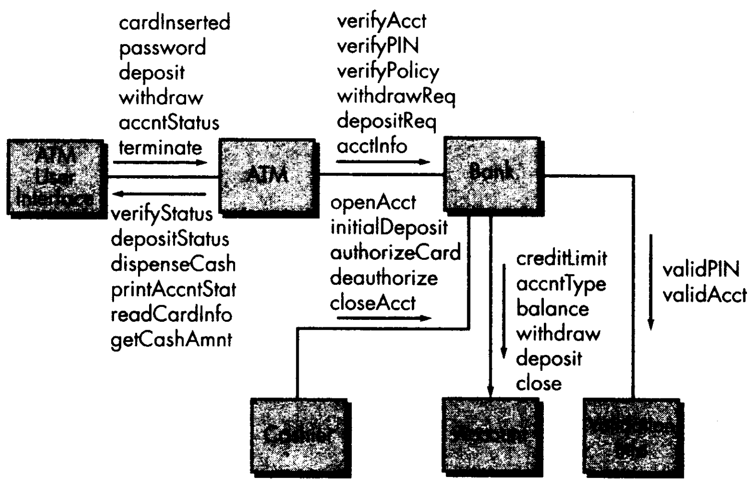
### 14.9.1 Multiple Class Testing

Kirani and Tsai [KIR94] suggest the following sequence of steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.

**Figure 14.11**

**Class collaboration diagram for banking application (adapted from [KIR94])**



3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

To illustrate [KIR94], consider a sequence of operations for the **Bank** class relative to an **ATM** class (Figure 14.11):

**verifyAcct•verifyPIN•[[verifyPolicy•withdrawReq]|depositReq|acctInfoREQ]$^n$**

A random test case for the **Bank** class might be

*Test case* $r_3$ = **verifyAcct•verifyPIN•depositReq**

In order to consider the collaborators involved in this test, the messages associated with each of the operations noted in test case $r_3$ is considered. **Bank** must collaborate with **ValidationInfo** to execute *verifyAcct()* and *verifyPIN()*. Bank must collaborate with **Account** to execute *depositReq()*. Hence, a new test case that exercises these collaborations is

*Test case* $r_4$ = **verifyAcctBank[validAcctValidationInfo]•verifyPINBank•[validPinValidationInfo]•depositReq•[depositaccount]**

The approach for multiple class partition testing is similar to the approach used for partition testing of individual classes. A single class is partitioned as discussed in Section 14.8.2. However, the test sequence is expanded to include those operations that are invoked via messages to collaborating classes. An alternative approach partitions tests based on the interfaces to a particular class. Referring to Figure 14.11, the **Bank** class receives messages from the **ATM** and **Cashier** classes. The methods within **Bank** can therefore be tested by partitioning them into those that serve **ATM** and those that serve **Cashier.** State-based partitioning (Section 14.8.2) can be used to refine the partitions further.

### 14.9.2 Tests Derived from Behavior Models

In Chapter 8, we discussed the use of the state diagram as a model that represents the dynamic behavior of a class. The state diagram for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it). Figure 14.12 [KIR94] illustrates a state diagram for the **Account** class discussed earlier. Referring to the figure, initial transitions move through the *empty acct* and *setup acct* states. The majority of all behavior for instances of the class occurs while in the *working acct* state. A final withdrawal and account closure cause the **Account** class to make transitions to the *nonworking acct* and *dead acct* states, respectively.