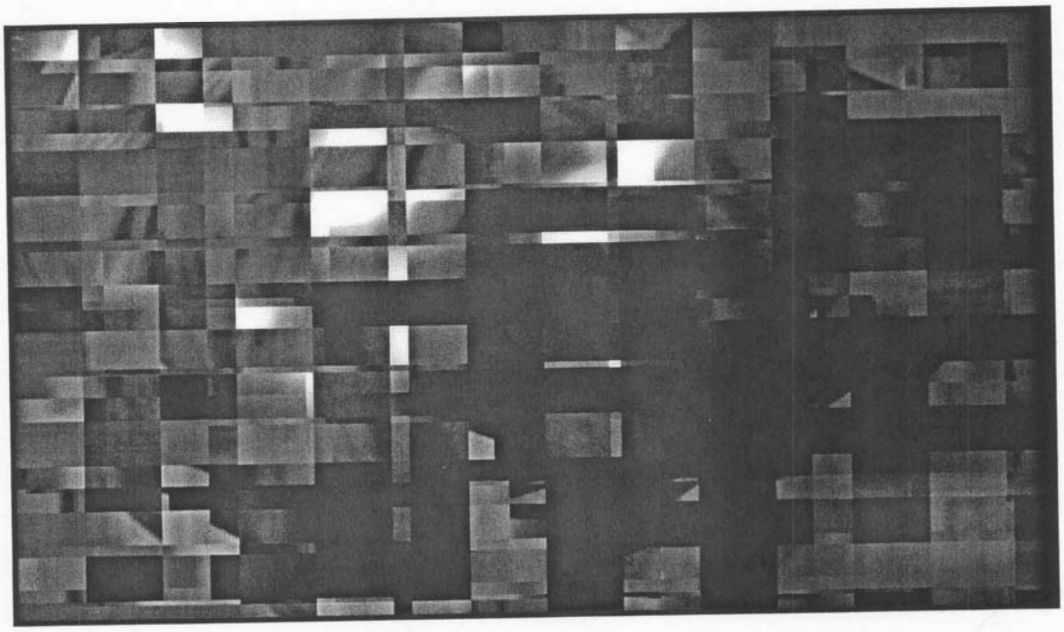
Software Engineering

A Practitioner's Approach

European Adaptation



Fifth Edition

Roger S. Pressman

adapted by

Darrel Ince

CHAPTER

18 SOFTWARE TESTING STRATEGIES

KEYWORDS

alpha/beta testing	482
criteria for completion	470
debugging	485
incremental strategies	476
ITG	468
integration testing	475
regression testing	478
smoke testing	480
system testing	483
unit testing	473
validation testing	481
V&V	467

A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in the successful construction of software. The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to promote reasonable planning and management tracking as the project progresses. Shooman [SHO83] discusses these issues:

> In many ways, testing is an individualistic process, and the number of different types of tests varies as much as the different development approaches. For many years, our only defence against programming errors was careful design and the native intelligence of the programmer. We are now in an era in which modern design techniques [and formal technical reviews] are helping us to reduce the number of initial errors that are inherent in the code. Similarly, different test methods are beginning to cluster themselves into several distinct approaches and philosophies.

QUICK LOOK

What is it? Designing effective test cases (Chapter 17) is important, but so is the strategy you use to execute them. Should you develop a formal plan for your tests? Should you test the entire program as a whole or run tests only on a small part of it? Should you rerun tests you've already conducted as you add new components to a large system? When should you involve the customer? These and many other questions are answered when you develop a software testing strategy.

Who does it? A strategy for software testing is developed by the project manager, software engineers, and testing specialists.

Why is it important? Testing often accounts for more project effort than any other software engineering activity. If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected. It would therefore seem reasonable to establish a systematic strategy for testing software.

What are the steps? Testing begins "in the small" and progresses "to the large." By this we mean that early testing focuses on a single component and applies white and black-box tests to uncover errors in program logic and function. After individual components are tested they must be integrated. Testing continues as the software is constructed. Finally, a series of high order tests are executed once the full program is operational. These tests are

QUICK LOOK

designed to uncover errors in requirements.

What is the work product? A *Test Specification* documents the software team's approach to testing by defining a plan that describes an overall strategy and a procedure that defines specific testing steps and the tests that will be conducted.

How do I ensure that I've done it right? By reviewing the *Test Specification* prior to testing, you can assess the completeness of test cases and testing tasks. An effective test plan and procedure will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

These "approaches and philosophies" are what we shall call strategy. In Chapter 17, the technology of software testing was presented.[1] In this chapter, we focus our attention on the strategy for software testing.

18.1 A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test case design techniques and testing methods—should be defined for the software process.

A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

WebRef

Useful information on software testing strategies is provided by the *Software Testing Newsletter* at: **www.ondaweb.com/sti/newsltr.htm**

- Testing begins at the component level[2] and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

A strategy for software testing must accommodate low level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high level tests that validate major system functions against customer requirements. A strategy must provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems must surface as early as possible.

[1] Testing for object-oriented systems is discussed in Chapter 23.

[2] For object-oriented systems, testing begins at the class or object level. See Chapter 23 for details.

18.1.1 Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). *Verification* refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm [BOE81] states this another way:

> *Verification:* "Are we building the product right?"
> *Validation:* "Are we building the right product?"

The definition of V&V encompasses many of the activities that we have referred to as software quality assurance (SQA).

XRef
SQA activities are discussed in detail in Chapter 8.

Verification and validation encompasses a wide array of SQA activities that include: formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing [WAL89]. Although testing plays an extremely important role in V&V, many other activities are also necessary

Quote:
"Testing is an unavoidable part of any responsible effort to develop a software system."
William Howden

Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective formal technical reviews and solid management and measurement all lead to quality that is confirmed during testing.

Miller [MIL77] relates [illegible] underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems."

18.1.2 Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error free, that it works according to customer requirements and that it will be completed on schedule and within budget. Each of these interests mitigate against thorough testing.

From a psychological point of view, software analysis and design (along with coding) are *constructive* tasks. The software engineer creates a computer program, its documentation, and related data structures. Like any builder, the software engineer is proud of the edifice that has been built and looks askance at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) *destructive.* So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than uncovering errors. Unfortunately, errors will be present. And if the software engineer doesn't find them, the customer will!

There are often a number of misconceptions that can be erroneously inferred from the above discussion: (1) that the developer of software should do no testing at all; (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly; (3) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

KEY POINT

An independent test group does not have the "conflict of interest" that builders of the software have.

The software developer is always responsible for testing the individual units (modules) of the program, ensuring that each performs the function for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete program structure. Only after the software architecture is complete does an independent test group become involved.

The role of an *independent test group* (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent test removes the conflict of interest that may otherwise be present. After all, personnel in the independent group team are paid to find errors.

ADVICE

If an ITG does not exist within your organization, you'll have to take its point of view. When you test, try to break the software.

However, the software engineer doesn't turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

The ITG is part of the software development project team in the sense that it becomes involved during the specification activity and stays involved (planning and specifying test procedures) throughout a large project. However, in many cases the ITG reports to the software quality assurance organization, thereby achieving a degree of independence that might not be possible if it were a part of the software engineering organization.

18.1.3 A Software Testing Strategy

The software engineering process may be viewed as a spiral illustrated in Figure 18.1. Initially, system engineering or information engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behaviour, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.

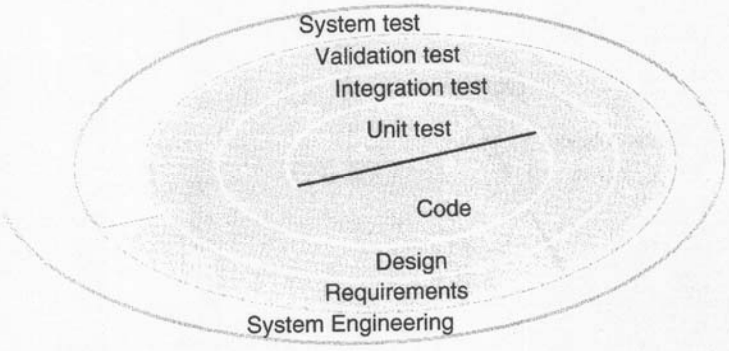


FIGURE 18.1
Testing strategy

? What is the overall strategy for software testing?

A strategy for software testing may also be viewed in the context of the spiral (Figure 18.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter *validation testing,* where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn.

XRef
Black-box and white-box testing techniques are discussed in Chapter 17.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 18.2. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name unit testing. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Black-box test case design techniques are the most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of high-order tests are conducted. Validation criteria (established during requirements analysis) must be tested. Validation testing provides final assurance that software meets all functional, behavioural, and performance requirements. Black-box testing techniques are used exclusively during validation.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once

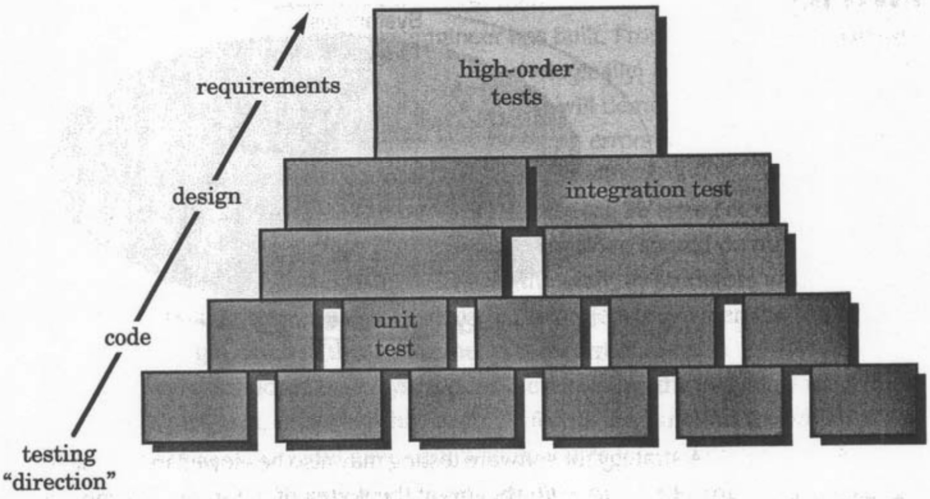


FIGURE 18.2 Software testing steps

validated, must be combined with other system elements (e.g., hardware, people, data bases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

18.1.4 Criteria for Completion of Testing

A classic question arises every time software testing is discussed: "When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

When are we done testing?

One response to the above question is: "You're never done testing, the burden simply shifts from you (the software engineer) to your customer." Every time the customer/user executes a computer program, the program is being tested on a new set of data. This sobering fact underlines the importance of other software quality assurance activities. Another response (somewhat cynical, but nonetheless accurate) is: "You're done testing when you run out of time or you run out of money."

Although few practitioners would argue with these responses, a software engineer needs more rigorous criteria for determining when sufficient testing has been conducted. Musa and Ackerman [MUS89] suggest a response that is based on statistical criteria: "No, we cannot be absolutely certain that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95 percent confidence that the probability of 1000 CPU hours of failure free operation in a probabilistically defined environment is at least 0.995."

Using statistical modelling and software reliability theory, models of software fail-

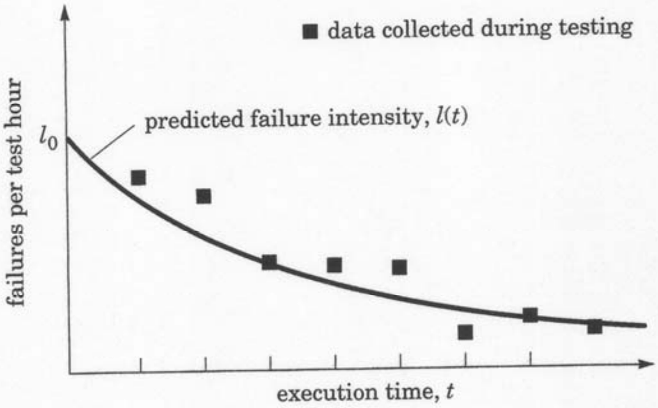


FIGURE 18.3 Failure intensity as a function of execution time

ures (uncovered during testing) as a function of execution time can be developed [MUS89]. A version of the failure model, called a *logarithmic Poisson execution-time model,* takes the form:

$$f(t) = (1/p) \ln [l_0 pt + 1)] \tag{18-1}$$

where $f(t)$ = cumulative number of failures that are expected to occur once the software has been tested for a certain amount of execution time, t,
l_0 = the initial software failure intensity (failures per unit time) at the beginning of testing,
p = the exponential reduction in failure intensity as errors are uncovered and repairs are made.

The instantaneous failure intensity, $l(t)$ can be derived by taking the derivative of $f(t)$,

$$l(t) = l_0 / (l_0 pt + 1) \tag{18-2}$$

Using the relationship noted in equation (18-2), testers can predict the drop-off of errors as testing progresses. The actual error intensity can be plotted against the predicted curve (Figure 18.3). If the actual data gathered during testing and the logarithmic Poisson execution time model are reasonably close to one another over a number of data points, the model can be used to predict total testing time required to achieve an acceptably low failure intensity.

By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?" There is little debate that further work remains to be done before quantitative rules for testing can be established, but the empirical approaches that currently exist are considerably better than raw intuition.

18.2 STRATEGIC ISSUES

Later in this chapter, we explore a systematic strategy for software testing. But even the best strategy will fail if a series of overriding issues are not addressed. Tom Gilb [GIL95] argues that the following issues must be addressed if a successful software testing strategy is to be implemented:

What guidelines lead to a successful testing strategy?

Specify product requirements in a quantifiable manner long before testing commences. Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability (Chapter 19). These should be specified in a way that is measurable so that testing results are unambiguous.

State testing objectives explicitly. The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, mean time to failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, test work-hours per regression test should all be stated within the test plan. [GIL95]

XRef

Use-cases describe a scenario for software use and are discussed in Chapter 11.

Understand the users of the software and develop a profile for each user category. Use-cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.

Develop a testing plan that emphasizes "rapid cycle testing." Gilb [GIL95] recommends that a software engineering team "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field "trialable," increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Quote:

"Testing only to end-user perceived requirements is like inspecting a building based on the work done by the interior decorator at the expense of the foundations, girders, and plumbing."

Boris Beizer

Build "robust" software that is designed to test itself. Software should be designed in a manner that uses antibugging (Section 18.3.1) techniques. That is, software should be capable of diagnosing certain classes of errors. In addition the design should accommodate automated testing and regression testing.

Use effective formal technical reviews as a filter prior to testing. Formal technical reviews (Chapter 8) can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high quality software.

Conduct formal technical reviews to assess the test strategy and test cases themselves. Formal technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.

Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

18.3 UNIT TESTING

Unit testing focuses verification effort on the smallest unit of software design—the software component or module.

18.3.1 Unit Test Considerations

Unit Testing

The tests that occur as part of unit tests are illustrated schematically in Figure 18.4. The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. An finally, all error handling paths are tested.

Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

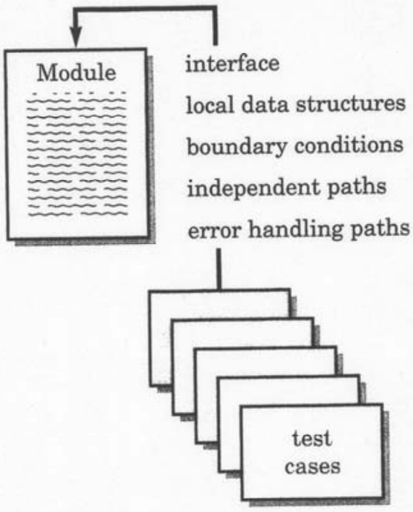


FIGURE 18.4
Unit test

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

What errors are commonly found during unit testing?

Among the more common errors in computation are: (1) misunderstood or incorrect arithmetic precedence; (2) mixed mode operations; (3) incorrect initialization; (4) precision inaccuracy; (5) incorrect symbolic representation of an expression. Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as: (1) comparison of different data types; (2) incorrect logical operators or precedence; (3) expectation of equality when precision error makes equality unlikely; (4) incorrect comparison of variables; (5) improper or non-existent loop termination; (6) failure to exit when divergent iteration is encountered, and (7) improperly modified loop variables.

Good design dictates that error conditions be anticipated and error-handling paths set up to re-route or cleanly terminate processing when an error does occur. Yourdon [YOU75] calls this approach *antibugging*. Unfortunately, there is a tendency to incorporate error handling into software and then never test it. A true story may serve to illustrate:

> A major interactive design system was developed under contract. In one transaction processing module, a practical joker placed the following error handling message after a series of conditional tests that invoked various control flow branches: ERROR! THERE IS NO WAY YOU CAN GET HERE. This "error message" was uncovered by a customer during user training!

ADVICE

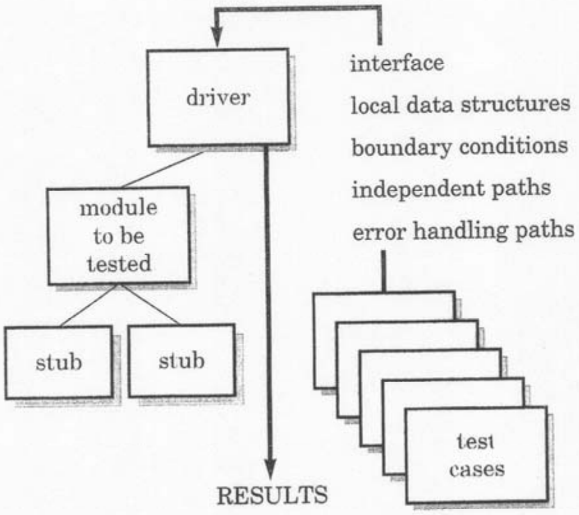
Be sure that you design tests to execute every error handling path. If you don't, the path may fail when it is invoked, exacerbating an already dicey situation.

Among the potential errors that should be tested when error handling is evaluated are:

1. Error description is unintelligible.
2. Error noted does not correspond to error encountered.
3. Error condition causes system intervention prior to error handling.
4. Exception-condition processing is incorrect.
5. Error description does not provide enough information to assist in the location of the cause of the error.

Boundary testing is the last (and probably most important) task of the unit test step. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed; when the *i*th repetition of a loop with *i* passes is invoked; when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

FIGURE 18.5
Unit test environment



18.3.2 Unit Test Procedures

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure 18.5. In most applications a *driver* is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

ADVICE

There are some situations in which you will not have the resources to do comprehensive unit testing. Select critical modules and those with high cyclomatic complexity and unit test only them.

Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

18.4 INTEGRATION TESTING[3]

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit tested: "If they all work individually, why do you doubt that

[3] Integration strategies for object-oriented systems are discussed in Chapter 23.

they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing. Data can be lost across an interface; one module can have an inadvertent, adverse affect on another; subfunctions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

ADVICE

Taking the big bang approach to integration is a lazy strategy that is doomed to failure. Integration testing should be conducted incrementally.

There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors are encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely, and a systematic test approach may be applied. In the sections that follow, a number of different incremental integration strategies are discussed.

18.4.1 Top-Down Integration

ADVICE

When you develop a detailed project schedule you have to consider the manner in which integration will occur so that components will be available when needed.

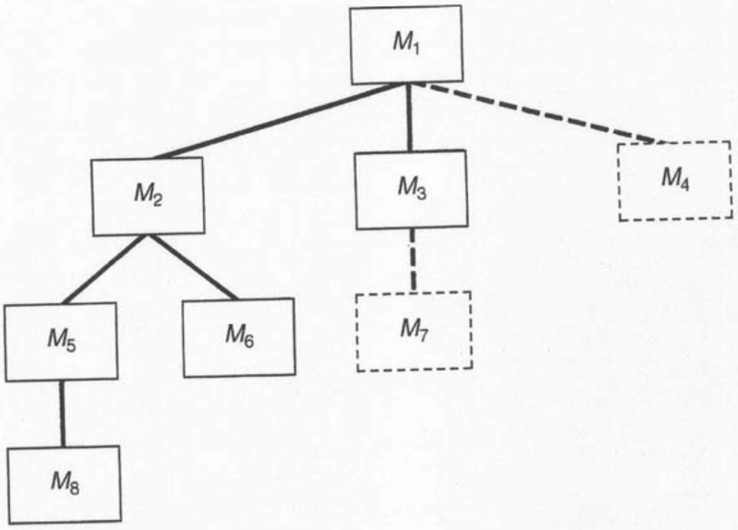
Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Referring to Figure 18.6, *depth-first integration* would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application specific characteristics. For example, selecting the left-hand path, components M_1, M_2, M_5 would be integrated first. Next, M_8 or (if necessary for proper functioning of M_2) M_6 would be integrated. Then, the central and right-hand control paths are built. *Breadth first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M_2, M_3 and M_4 (a replacement for stub, S_4) would be integrated first. The next control level, M_5, M_6, and so on, follows.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

FIGURE 18.6
Top-down integration



? What are the steps for top-down integration?

2. Depending on the integration approach selected (i.e., depth or breadth-first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing (Section 18.4.3) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

XRef
Factoring is important for certain architectural styles. See Chapter 14 for additional details.

The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure (Chapter 14) in which a complex series of interactive inputs are requested, acquired and validated via an incoming path. The incoming path may be integrated in a top down manner. All input processing (for subsequent transaction dispatching) may be demonstrated before other elements of the structure have been integrated. Early demonstration of functional capability is a confidence builder for both the developer and the customer.

Top down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low level modules at the beginning of top down testing; therefore, no significant data can

What problems may be encountered when the top-down integration strategy is chosen?

flow upward in the program structure. The tester is left with three choices: (1) delay many tests until stubs are replaced with actual modules, (2) develop stubs that perform limited functions that simulate the actual module or (3) integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) causes us to lose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top down approach. The second approach is workable, but can lead to significant overhead, as stubs become more and more complex. The third approach, called bottom-up testing is discussed in the next section.

18.4.2 Bottom-Up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

What are the steps for bottom-up integration?

1. Low level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

KEY POINT Bottom-up integration eliminates the need for complex stubs.

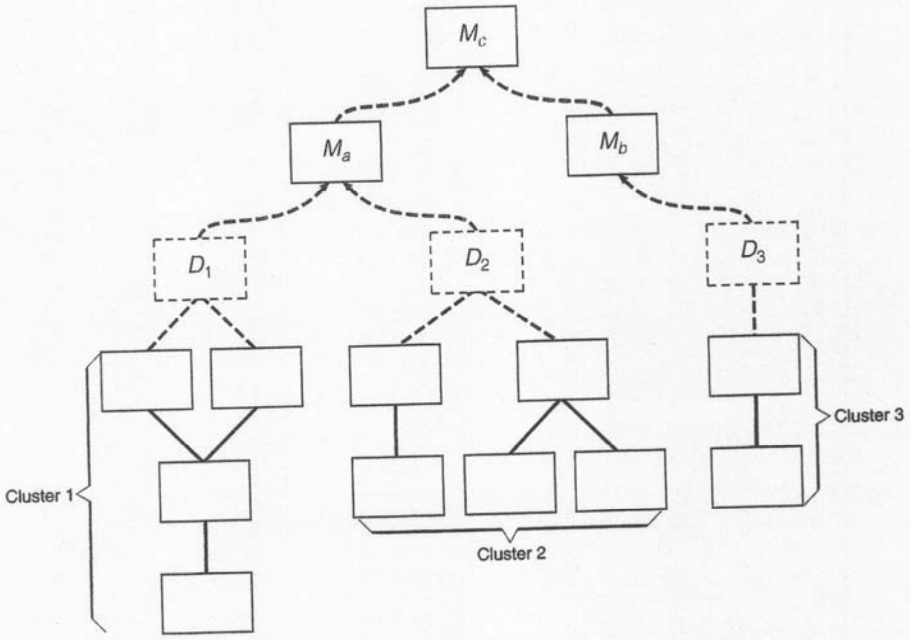
Integration follows the pattern illustrated in Figure 18.7. Components are combined to form clusters 1, 2 and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

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.

18.4.3 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

FIGURE 18.7
Bottom-up integration



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 behaviour or additional errors.

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 modules).

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.

18.4.4 Smoke Testing

Smoke testing is an integration testing approach that is commonly used when "shrink-wrapped" 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:

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

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 are 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.

Quote:

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

Jim McCarthy

- *The quality of the end-product is improved.* Because it is a construction (integration) oriented approach, smoke testing is likely to uncover both functional errors and also architectural and component-level design defects. If these defects 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.

18.4.5 Comments on Integration Testing

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 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.

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

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 for upper levels of the program structure, coupled with a bottom-up for subordinate levels may be the best compromise.

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 (cyclomatic complexity may be used as an indicator), or (4) has definite performance requirements. Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

18.5 VALIDATION TESTING

At the culmination of integration testing, software is completely assembled as a package; interfacing errors have been uncovered and corrected, and a final series of software tests—*validation testing*—may begin. Validation can be defined in many ways,

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.

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 (Chapter 12) 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.

18.5.1 Validation Test Criteria

Software validation is achieved through a series of black-box 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 that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied; all behavioural characteristics are achieved; all performance requirements are attained; documentation is correct and human-engineered, 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 characteristics conform to specification and are 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.

18.5.2 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 catalogued, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometime called an *audit,* has been discussed in more detail in Chapter 9.

18.5.3 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.

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 a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more customer sites by the end user of the software. 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 customer 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 test, software engineers make modifications and then prepare for release of the software product to the entire customer base.

18.6 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.

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.

Quote:

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

Ed Yourdon

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.

18.6.1 Recovery Testing

WebRef

Extensive information of software testing and related quality issues can be obtained at **www.stqe.net**

Many computer based systems must recover from faults and resume processing within a prespecified time. In some cases, a system must be fault tolerant, i.e., 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), re-initialization, checkpointing mechanisms, data recovery and restart are each evaluated for correctness. If recovery requires human intervention, the mean time to repair (MTTR) is evaluated to determine whether it is within acceptable limits.

18.6.2 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 attempts to verify 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 defences 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.

Quote:

"If you're trying to find true system bugs and have never subjected your software to a real stress test, then it is high time you started."

Boris Beizer

18.6.3 Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted 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 10 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 thrashing in a virtual operating system are designed; (5) test cases that may cause excessive hunting for disk resident data are designed. Essentially, the tester attempts to break the program.

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.

18.6.4 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 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 white-box 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.

18.7 THE ART OF DEBUGGING

Software testing is a process 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.

WebRef

BugNet tracks security problems and bugs in PC-based software and provides useful informat on debugging topics: **www.bugnet.com**

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process 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.

18.7.1 The Debugging Process

Debugging is not testing, but always occurs as a consequence of testing.[4] Referring to Figure 18.8, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual is encountered. In many cases, the non-corresponding data is a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will always have one of two outcomes: (1) the cause will be found, corrected and removed, or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate her 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 that software technology. However, a few characteristics of bugs provide some clues:

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 program structures (Chapter 13) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non-errors (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].

Quote:

"The variety within all computer programs that must be diagnosed [for debugging] is probably greater than the variety within all other examples of systems that are regularly diagnosed."

John Gould

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 sometimes forces a software developer to fix one error and at the same time introduce two more.

[4] In making this 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!

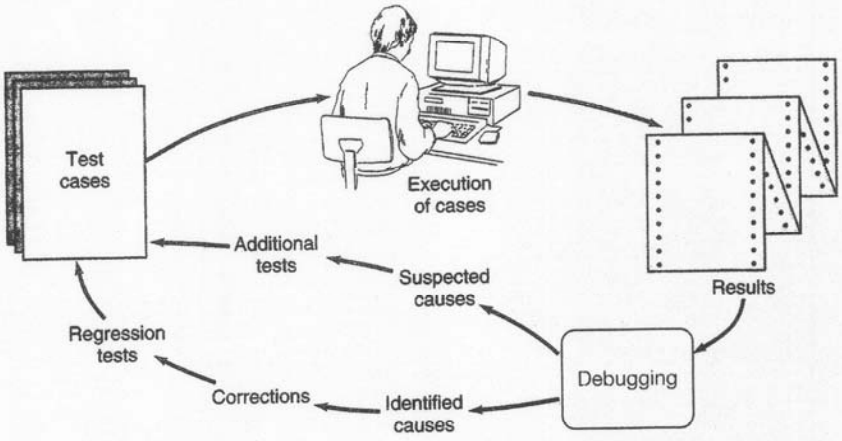


FIGURE 18.8 The debugging process

18.7.2 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 educational and experiential background.

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.

18.7.3 Debugging Approaches

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 neighbourhood 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 categories for debugging approaches may be proposed [MYE79]: (1) brute force, (2) backtracking, and (3) cause elimination.

ADVICE

Set a time limit, say 2 hours, on the amount of time you spend trying to debug a problem on your own. After that, get help!

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 WRITE 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 above 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.

CASE Tools
Testing and Debugging

Each of the above debugging approaches can be supplemented with debugging tools. We can apply a wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test case generators, memory dumps, and cross reference maps. However, tools are not a substitute for careful evaluation based on a complete software design document and clear source code.

Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! Each of us can recall puzzling for hours or days over a persistent bug. A colleague wanders by and in desperation we explain the problem and throw open the listing. Instantaneously (it seems), the cause of the error is uncovered. Smiling smugly, our colleague wanders off. A fresh viewpoint, unclouded by hours of frustration, can do wonders. A final maxim for debugging might be: "When all else fails, get help!"

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 defect 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 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 8). 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.

18.8 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 fulfil 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 module and incorporation of modules into a program structure. 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

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

[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, Inc., San Francisco, January, 1995.

[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.

[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

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

18.2. 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?

18.3. Is it always possible to develop a strategy for testing software that uses the sequence of testing steps described in Section 18.1.3? What are possible complications that might arise for embedded systems?

18.4. If you could only select three test case design methods to apply during unit testing, what would they be and why?

18.5. Why is it that a highly-coupled module is difficult to unit test?

18.6. Develop an integration testing strategy for any one of the systems implemented in Problems 16.4 through 16.11. Define test phases, note the order of integration, specify additional test software and justify your order of integration. Assume that all modules (or classes) have been unit tested and are available. Note: it may be necessary to do a bit of design work first.

18.7. How can project scheduling affect integration testing?

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

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

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

18.11. 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.

FURTHER READINGS AND INFORMATION RESOURCES

Books by Black (*Managing the Testing Process,* Microsoft Press, 1999), Dustin, Rashka and Paul (*Test Process Improvement: Step-By-Step Guide to Structured Testing,* Addison-Wesley, 1999), Perry (*Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach,* Dorset House, 1997), and Kit and Finzi (*Software Testing in the Real World: Improving the Process,* Addison-Wesley, 1995) address software testing strategies.

Kaner, Nguyen, and Falk (*Testing Computer Software,* Wiley, 1999), Hutcheson (*Software Testing Methods and Metrics: The Most Important Tests* McGraw Hill, 1997), Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing,* Prentice Hall, 1995), Jorgensen (*Software Testing: A Craftsman's Approach,* CRC Press, 1995) present treatments of the subject that consider testing methods and strategies.

In addition, older books by Evans (*Productive Software Test Management,* Wiley-Interscience, 1984), Hetzel (*The Complete Guide to Software Testing,* QED Information Sciences, 1984), Beizer [BEI84], Ould and Unwin (*Testing in Software Development,* Cambridge University Press, 1986), Marks (*Testing Very Big Systems,* McGraw-Hill, 1992, and Kaner et al (*Testing Computer Software,* second edition, Van Nostrand-Reinhold, 1993) delineate the steps of an effective testing strategy, provide a set of techniques and guidelines and suggest procedures for controlled and tracking the testing process. Hutcheson (*Software Testing Methods and Metrics,* McGraw-Hill, 1996) presents testing methods and strategies but also provides a detailed discussion of how measurement can be used to achieve efficient testing.

Guidelines for debugging are contained in a book by Dunn (*Software Defect Removal,* McGraw-Hill, 1984). Beizer [BEI84] presents an interesting "taxonomy of bugs" that can lead to effective methods for test planning. McConnell (*Code Complete,* Microsoft Press, 1993) presents pragmatic advice on unit and integration testing as well as debugging.

A wide variety of information sources on software testing and related subjects are available on the internet. An up-to-date list of world wide web references that are relevant to testing concepts, methods and strategies can be found at **http://www.pressman5.com**