

A complete discussion of user interface evaluation methods is best left to books dedicated to the subject. For further information, see [LEA88], [MAN97], and [HAC98].

12.6 SUMMARY

The user interface is arguably the most important element of a computer-based system or product. If the interface is poorly designed, the user's ability to tap the computational power of an application may be severely hindered. In fact, a weak interface may cause an otherwise well-designed and solidly implemented application to fail.

Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user's memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted.

The development of a user interface begins with a series of analysis tasks. These include user identification, task, and environmental analysis/modeling. User analysis defines the profiles of various end-users and applies information gathered from a variety of business and technical sources. Task analysis defines user tasks and actions using either an elaborative or object-oriented approach, applying use-cases, task and object elaboration, workflow analysis, and hierarchical task representations to fully understand the human-computer interaction. Environmental analysis identifies the physical and social structures in which the interface must operate.

Once tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. This provides a basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. A variety of implementation tools are used to build a prototype for evaluation by the user.

REFERENCES

- [APP03] Apple Computer, *People with Special Needs*, 2003, available at <http://www.apple.com/disability/>.
- [BOR01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley, 2001.
- [CON95] Constantine, L., "What DO Users Want? Engineering Usability in Software," *Windows Tech Journal*, December, 1995, available from <http://www.forUse.com>.
- [DON99] Donahue, G., S. Weinschenck, and J. Nowicki, "Usability Is Good Business," Compuware Corp., July, 1999, available from <http://www.compuware.com>.
- [DUY02] vanDuyne, D., J. Landay, and J. Hong, *The Design of Sites*, Addison-Wesley, 2002.
- [HAC98] Hackos, J., and J. Redish, *User and Task Analysis for Interface Design*, Wiley, 1998.
- [IBM03] IBM, *Overview of Software Globalization*, 2003, available at <http://oss.software.ibm.com/icu/userguide/i18n.html>.

- [LEA88] Lea, M., "Evaluating User Interface Designs," *User Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.
- [MAN97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.
- [MIC03] *Microsoft Accessibility Technology for Everyone*, 2003, available at <http://www.microsoft.com/enable/>.
- [MON84] Monk, A., (ed.), *Fundamentals of Human-Computer Interaction*, Academic Press, 1984.
- [MOR81] Moran, T. P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *Intl. Journal of Man-Machine Studies*, vol. 15, pp. 3-50.
- [NIE00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.
- [NOR86] Norman, D. A., "Cognitive Engineering," in *User Centered Systems Design*, Lawrence Erlbaum Associates, 1986.
- [RUB88] Rubin, T., *User Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.
- [SHN90] Shneiderman, B., *Designing the User Interface*, 3rd ed., Addison-Wesley, 1990.
- [TID99] Tidwell, J., "Common Ground: A Pattern Language for HCI Design," available at http://www.mit.edu/~jtidwell/interaction_patterns.html, May 1999.
- [TID02] Tidwell, J., "IU Patterns and Techniques," available at <http://time-tripper.com/uipatterns/index.html>, May 2002.
- [UNI03] Unicode, Inc., *The Unicode Home Page*, 2003, available at <http://www.unicode.org/>.
- [W3C03] World Wide Web Consortium, *Web Content Accessibility Guidelines*, 2003, available at <http://www.w3.org/TR/2003/WD-WCAG20-20030624/>.
- [WEL01] vanWelie, M., "Interaction Design Patterns," available at <http://www.welie.com/patterns/>, 2001.

PROBLEMS AND POINTS TO PONDER

- 12.1. Provide a few examples that illustrate why response time variability can be an issue.
- 12.2. Develop two additional design principles that "reduce the user's memory load."
- 12.3. Add at least five additional questions to the list developed for content analysis in Section 12.3.3.
- 12.4. Develop two additional design principles that "make the interface consistent."
- 12.5. You have been asked to develop a Web-based home banking system. Develop a user model, design model, mental model, and an implementation model.
- 12.6. Develop a set of screen layouts with a definition of major and minor menu items for the system in Problem 12.5.
- 12.7. Develop two additional design principles that "place the user in control."
- 12.8. Perform a detailed task analysis for the system in Problem 12.5. Use either an elaborative or object-oriented approach.
- 12.9. Develop a set of screen layouts with a definition of major and minor menu items for the *SafeHome* system. You may elect to take a different approach than the one shown for the screen layout in Figure 12.3.
- 12.10. Continuing Problem 12.8, define interface objects and actions for the application. Identify each object type.
- 12.11. Describe your approach to user help facilities for the task analysis you have performed as part of Problem 12.5.
- 12.12. Describe the best and worst interfaces that you have ever worked with and critique them relative to the concepts introduced in this chapter.

12.13. Develop an approach that would automatically integrate error messages and a user help facility. That is, the system would automatically recognize the error type and provide a help window with suggestions for correcting it. Perform a reasonably complete software design that considers appropriate data structures and algorithms.

12.14. Develop an interface evaluation questionnaire that contains 20 generic questions that would apply to most interfaces. Have 10 classmates complete the questionnaire for an interactive system that you all use. Summarize the results and report them to your class.

Although his book is not specifically about human/computer interfaces, much of what Donald Norman (*The Design of Everyday Things*, reissue edition, Currency/Doubleday, 1990) has to say about the psychology of effective design applies to the user interface. It is recommended reading for anyone who is serious about doing high-quality interface design.

Graphical user interfaces are ubiquitous in the modern world of computing. Whether it is used for an ATM, a mobile phone, a PDA, a Web site, or a business application, the user interface provides a window into the software. It is for this reason that books addressed to interface design abound. Galitz (*The Essential Guide to User Interface Design*, Wiley, 2002), Cooper (*About Face 2.0: The Essentials of User Interface Design*, IDG Books, 2003), Beyer and Holtzblatt (*Contextual Design: A Customer Centered Approach to Systems Design*, Morgan-Kaufmann, 2002), Raskin (*The Humane Interface*, Addison-Wesley, 2000), Constantine and Lockwood (*Software for Use*, ACM Press, 1999), Mayhew (*The Usability Engineering Lifecycle*, Morgan-Kaufmann, 1999) all discuss usability, user interface concepts, principles, and design techniques and contain many useful examples.

Johnson (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kaufmann, 2000) provides useful guidance for those that learn more effectively by examining counter-examples. An enjoyable book by Cooper (*The Inmates Are Running the Asylum*, Sams Publishing, 1999) discusses why high-tech products drive us crazy and how to design ones that don't.

Task analysis and modeling are pivotal interface design activities. Hackos and Redish [HAC98] have written a book dedicated to these subjects and provide a detailed method for approaching task analysis. Wood (*User Interface Design: Bridging the Gap from User Requirements to Design*, CRC Press, 1997) considers the analysis activity for interfaces and the transition to design tasks.

The evaluation activity focuses on usability. Books by Rubin (*Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, Wiley, 1994) and Nielsen (*Usability Inspection Methods*, Wiley, 1994) address the topic in considerable detail.

In a unique book that may be of considerable interest to product designers, Murphy (*Front Panel: Designing Software for Embedded User Interfaces*, R&D Books, 1998) provides detailed guidance for the design of interfaces for embedded systems and addresses safety hazards inherent in controls, handling heavy machinery, and interfaces for medical or transport systems. Interface design for embedded products is also discussed by Garrett (*Advanced Instrumentation and Computer I/O Design: Real-Time System Computer Interface Engineering*, IEEE, 1994).

A wide variety of information sources on user interface design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to user interface design can be found at the SEPA Web site:

<http://www.mhhe.com/pressman>.

CHAPTER

13

TESTING STRATEGIES

KEY CONCEPTS

- alpha/beta testing
- debugging
- completion criteria
- conventional strategy
- integration testing
- ITG
- OO strategy
- regression testing
- smoke testing
- system testing
- test specification
- unit testing
- V&V
- validation testing

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:

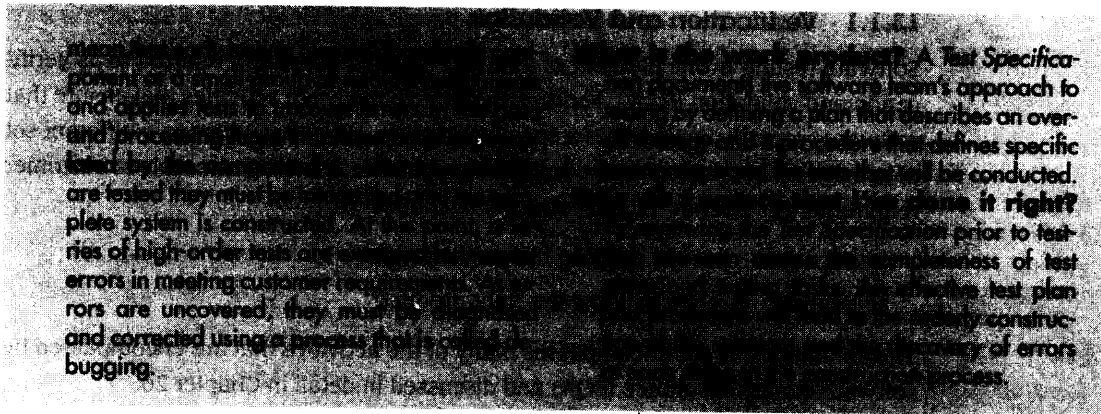
In many ways, testing is an individualistic process, and the number of different types of tests varies as much as the different development approaches. For many years, our only defense against programming errors was careful design and the native intelligence of the programmer. We are now in an era in 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.

These "approaches and philosophies" are what we shall call *strategy*. In Chapter 14, the technology of software testing is presented. In this chapter, we focus our attention on the strategy for software testing.

QUICK LOOK

What is the purpose of testing? Should you develop a formal plan for 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 errors creep through undetected. It would therefore seem reasonable to establish a systematic strategy for testing software. What does the strategy? Testing begins "in the small" and progresses "to the large." By this we



13.1 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 resources for software testing can be found at www.mtsu.edu/~storm/.

- To perform effective testing, a software team should conduct effective formal technical reviews (Chapter 26). By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level 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.

13.1.1 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.¹ 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 are encompassed by software quality assurance (SQA) and discussed in detail in Chapter 26.

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, usability testing, qualification testing, and installation testing [WAL89]. Although testing plays an extremely important role in V&V, many other activities are also necessary.

Testing is the last part of any responsible effort to develop a software system.



Don't get sloppy and view testing as a "safety net" that will catch all errors that occurred because of weak software engineering practices. It won't. Stress quality and error detection throughout the software process.

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 software testing to quality assurance by stating that "the 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."

13.1.2 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

¹ It should be noted that there is a strong divergence of opinion about what types of testing constitute "validation." Some people believe that *all* testing is verification and that validation is conducted when requirements are reviewed and approved, and later, by the user when the system is operational. Other people view unit and integration testing (Sections 13.3.1 and 13.3.2) as verification and higher-order testing (discussed later in this chapter) as validation.

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.

“The greatest occupational hazard of programming; testing is the treatment.”
—Steve Jobs

From a psychological point of view, software analysis and design (along with coding) are constructive tasks. The software engineer analyzes, models, and then creates a computer program and its documentation. 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 preceding 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.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior 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 software architecture. 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 testing removes the conflict of interest that may otherwise be present. After all, ITG personnel are paid to find errors.

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.

KEY POINT

An independent test group does not have the “conflict of interest” that builders of the software might experience.

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.

“The biggest mistake that people make is thinking that the testing team is responsible for assuring quality.”
—Steve Jobs

The ITG is part of the software development project team in the sense that it becomes involved during analysis and design 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.

13.1.3 A Software Testing Strategy for Conventional Software Architectures

The software process may be viewed as the spiral illustrated in Figure 13.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, 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.

A strategy for software testing may also be viewed in the context of the spiral (Figure 13.1). *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.

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 13.2. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name

What is the overall strategy for software testing?

WebRef

Useful resources for software testers can be found at www.SQAtester.com.

FIGURE 13.1

Testing strategy

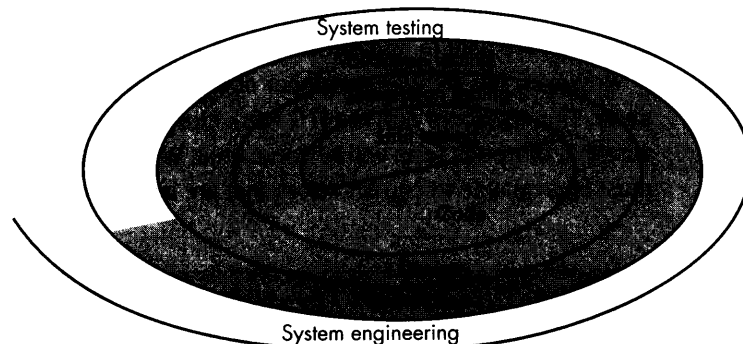
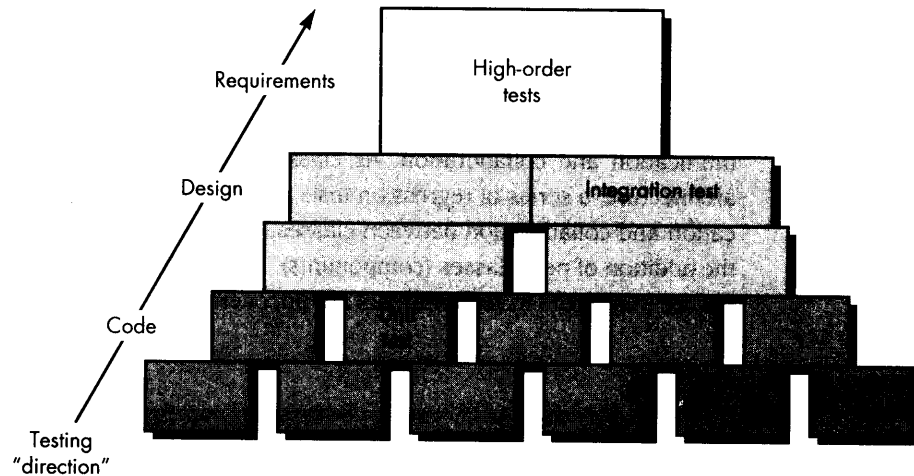


FIGURE 13.2
Software testing steps



unit testing. Unit testing makes heavy use of testing techniques that exercise specific paths in a component'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. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths 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 evaluated. Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

13.1.4 A Software Testing Strategy for Object-Oriented Architectures

The testing of object-oriented systems presents a different set of challenges for the software engineer. The definition of testing must be broadened to include error discovery techniques (e.g., formal technical reviews) that are applied to analysis and design models. The completeness and consistency of object-oriented representations must be assessed as they are built. Unit testing loses some of its meaning, and integration strategies change significantly. In summary, both testing strategies and testing tactics (Chapter 14) must account for the unique characteristics of object-oriented software.

KEY POINT

Like conventional testing, OO testing begins “in the small.” However, in most cases, the smallest element tested is a class or package of collaborating classes.

The overall strategy for object-oriented software is identical in philosophy to the one applied for conventional architectures, but differs in approach. We begin with “testing in the small” and work outward toward “testing in the large.” However, our focus when “testing in the small” changes from an individual module (the conventional view) to a class that encompasses attributes and operations and implies communication and collaboration. As classes are integrated into an object-oriented architecture, a series of regression tests are run to uncover errors due to communication and collaboration between classes (components) and side effects caused by the addition of new classes (components). Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

SAFEHOME



Preparing for Testing

Ed: Welcome. Doug Miller's office, as you can see, is in the middle of construction of a new building. We're a software engineering firm, and I'm Vinod Shukla—members of Doug's engineering team.

Jamie: I haven't spent enough time

... I've all been just a little busy. And I haven't been thinking about it ... in fact, more

... we're all overloaded, but I haven't had time.

Ed: The idea of designing unit tests before I start programming, so that's what I've been doing. It's a pretty big file of tests to run when the development is complete.

Jamie: I've heard about Programming [an agile software engineering concept, see Chapter 4] concept, no?

Ed: It is. Even though we're not using Agile Programming per se, we decided that was a good idea to design unit tests before we start programming. The design gives us all of the information we need.

Jamie: I've been doing the same thing.

Vinod: And I've taken on the role of testing every time one of the guys passes it. I'll integrate it and run a series of regression tests on a partially integrated program. I've been using a set of appropriate tests for each function.

Doug (to Vinod): How often will you be testing?

Vinod: Every day ... until the system is complete. Well, I mean until the software increment we're going to deliver is integrated.

Doug: You guys are way ahead of me.

Vinod (laughing): Anticipate it, Boss. It's the software biz, Boss.

13.1.5 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.

One response to the 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. 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 modeling and software reliability theory, models of software failures (uncovered during testing) as a function of execution time can be developed (e.g., see [MUS89], [SIN99] or [IEE01]).

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.



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 software 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 15). 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, and test work-hours per regression test all should be stated within the test plan [GIL95].

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.

Build “robust” software that is designed to test itself. Software should be designed in a manner that uses antialiasing (Section 13.3.1) techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.

WebRef

An excellent list of testing resources can be found at www.ie.com/~wuzmo/qn/.

Use effective formal technical reviews as a filter prior to testing. Formal technical reviews (Chapter 26) 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.

“Testing only to end user requirements is like inspecting a building based on the work done by the interior designer at the expense of the foundations, girders, and plumbing.”

Boris Becker

13.3 TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

There are many strategies that can be used to test software. At one extreme, a software team could wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors. This approach, although appealing, simply does not work. It will result in buggy software that disappoints the customer and end-user. At the other extreme, a software engineer could conduct tests on a daily basis, whenever any part of the system is constructed. This approach, although less appealing to many, can be very effective. Unfortunately, most software developers hesitate to use it. What to do?

A testing strategy that is chosen by most software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

13.3.1 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of

the module. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

Unit Test Considerations. The tests that occur as part of unit tests are illustrated schematically in Figure 13.3. The module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. 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. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error handling paths are tested.

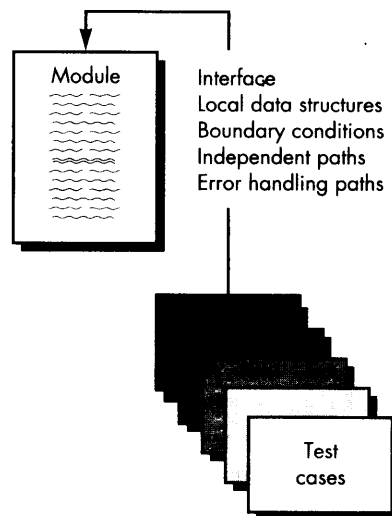
Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

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. Among the more common errors in computation are (1) misunderstood or incorrect arithmetic precedence, (2) mixed mode operations, (3) incorrect initialization, (4) precision inaccuracy, and (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 com-

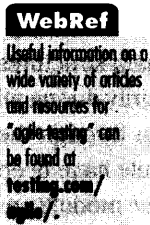
What errors are commonly found during unit testing?

FIGURE 13.3

Unit test



parison). 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 nonexistent loop termination, (6) failure to exit when divergent iteration is encountered, and (7) improperly modified loop variables.



Boundary testing is one of the most important unit testing tasks. 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.

Good design dictates that error conditions be anticipated and error-handling paths set up to reroute 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:



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.

A computer-aided 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!

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 operating system intervention prior to error handling; (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

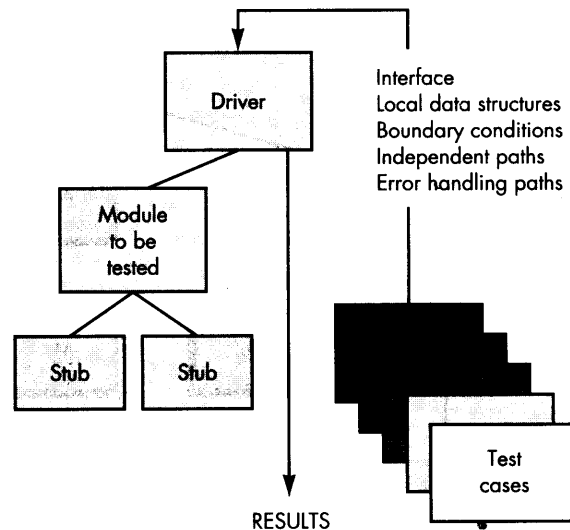
Unit test procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can be performed before coding begins (a preferred agile approach) or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

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 13.4. 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 to (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing.



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 those.

FIGURE 13.4
Unit test environment



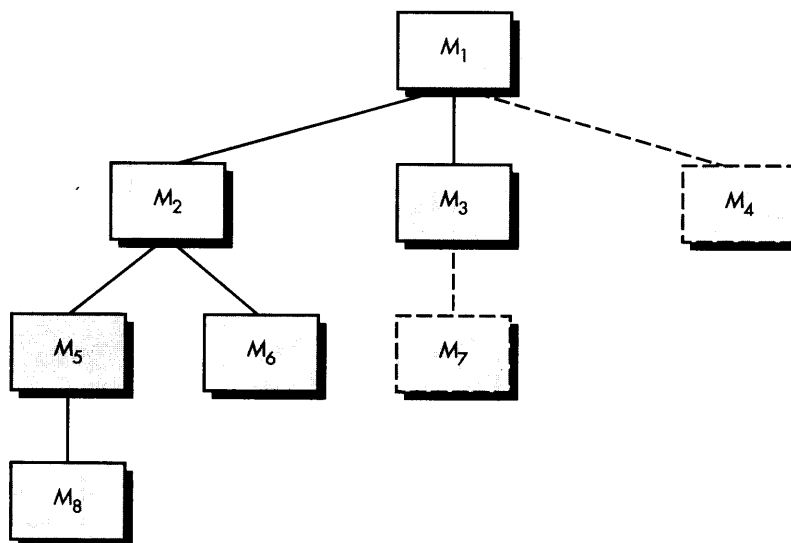
Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately 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.

13.3.2 Integration Testing

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 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 software architecture 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.

FIGURE 13.5**Top-down integration**

ADVICE
Taking the "big bang" approach to integration is a lazy strategy that is doomed to failure. Integrate incrementally, testing as you go.

There is often a tendency to attempt nonincremental 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 is 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 paragraphs that follow, a number of different incremental integration strategies are discussed.



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

Top-down integration. *Top-down integration testing* is an incremental approach to construction of the software architecture. 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 13.5, *depth-first integration* integrates all components on a major control path of the program 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 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:

 What are the steps for top-down integration?

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.
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 (discussed later in this section) 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.

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 10) in which a complex series of interactive inputs is 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.

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

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 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.

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.

Integration follows the pattern illustrated in Figure 13.6. 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.

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

FIGURE 13.6

Bottom-up integration

