CHAPTER

17 Software Testing Strategies

Key Concepts

alpha test	469
beta test	469
class testing	466
configuration review	468
debugging	473
deployment testing	472
independent test group	452

A strategy for software testing 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 encourage 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

Quick Look

What is it? Software is tested to uncover errors that were made inadvertently as it was designed and constructed. But how do you conduct the tests? 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 action. 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 I mean that early testing focuses on a single component or a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements. As errors are uncovered, they must be diagnosed and corrected using a process that is called debugging.

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 types of 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.

integration testing 459
regression testing 467
system testing ...470
unit testing 456
validation testing 467
V&V 450

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 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 I call *strategy*—the topic to be presented in this chapter. In Chapters 18 through 20, the testing methods and techniques that implement the strategy are presented.

17.1 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 you 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 you with a template for testing and all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews (Chapter 15). 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 for different software engineering approaches and 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.

WebRef
Useful resources for software testing can be found at **www.mtsu.edu/~storm/**.

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 should 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 should surface as early as possible.

17.1.1 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 tasks that ensure that software correctly implements a specific function. *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

uote:

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

William Howden

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

The definition of V&V encompasses many software quality assurance activities (Chapter 16).[1]

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

ADVICE

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 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."

17.1.2 Organizing for Software Testing

uote:

"Optimism is the occupational hazard of programming; testing is the treatment."

Kent Beck

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 analyzes, models, and then creates a computer program and its documentation. Like any builder, the software

[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 17.3.1 and 17.3.2) as verification and higher-order testing (Sections 17.6 and 17.7) as validation.

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 you might infer erroneously 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.

KEY POINT

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

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, you don'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 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.

Quote:

"The first mistake that people make is thinking that the testing team is responsible for assuring quality."

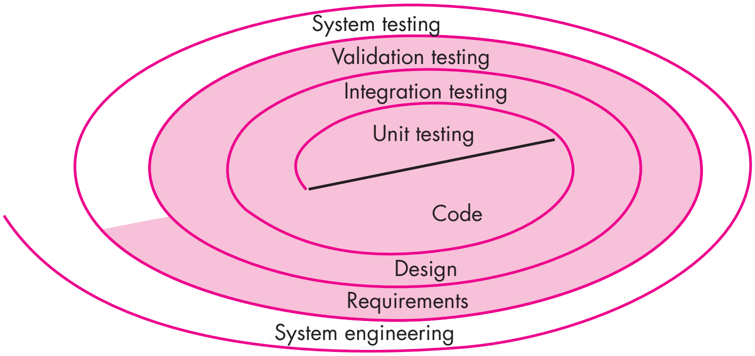
Brian Marick

17.1.3 Software Testing Strategy—The Big Picture

The software process may be viewed as the spiral illustrated in Figure 17.1. 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, you come to design and finally to coding. To develop computer software, you spiral inward (counterclockwise) along streamlines that decrease the level of abstraction on each turn.

FIGURE 17.1

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 17.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) 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, you encounter *validation testing,* where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at *system testing,* where the software and other system elements are tested as a whole. To test computer software, you spiral out in a clockwise direction along streamlines that broaden the scope of testing with each turn.

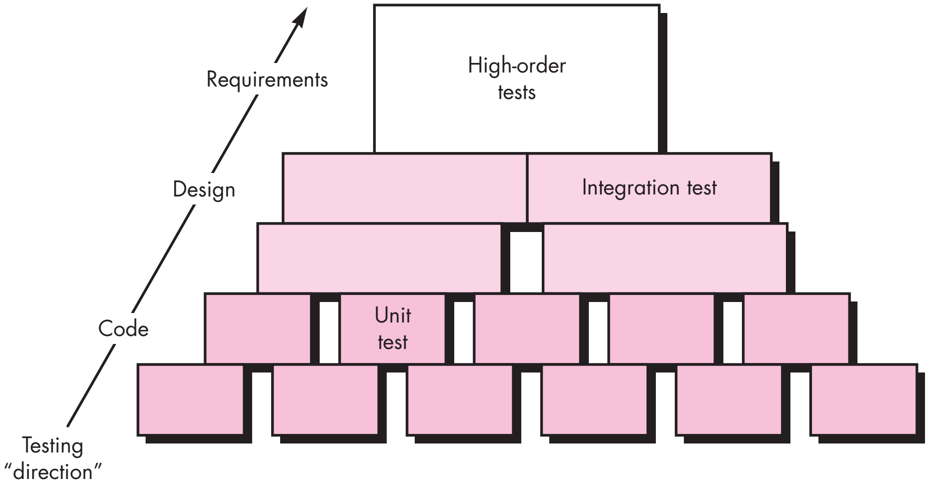
WebRef

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

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 17.2. Initially, tests focus on each

FIGURE 17.2

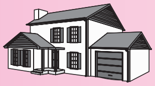
Software testing steps



component individually, ensuring that it functions properly as a unit. Hence, the name *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* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all informational, 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.

SafeHome



Preparing for Testing

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

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

The conversation:

Doug: It seems to me that we haven't spent enough time talking about testing.

Vinod: True, but we've all been just a little busy. And besides, we have been thinking about it . . . in fact, more than thinking.

Doug (smiling): I know . . . we're all overloaded, but we've still got to think down the line.

Shakira: I like the idea of designing unit tests before I begin coding any of my components, so that's what I've been trying to do. I have a pretty big file of tests to run once code for my components is complete.

Doug: That's an Extreme Programming [an agile software development process, see Chapter 3] concept, no?

Ed: It is. Even though we're not using Extreme Programming per se, we decided that it'd be a good idea to design unit tests before we build the component—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 the integrator, so every time one of the guys passes a component to me, I'll integrate it and run a series of regression tests on the partially integrated program. I've been working to design a set of appropriate tests for each function in the system.

Doug (to Vinod): How often will you run the tests?

Vinod: Every day . . . until the system is integrated . . . well, I mean until the software increment we plan to deliver is integrated.

Doug: You guys are way ahead of me!

Vinod (laughing): Anticipation is everything in the software biz, Boss.

17.1.4 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 finished testing?

One response to the question is: "You're never done testing; the burden simply shifts from you (the software engineer) to the end user." Every time the 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, you need more rigorous criteria for determining when sufficient testing has been conducted. The *cleanroom software engineering* approach (Chapter 21) suggests statistical use techniques [Kel00] that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population. Others (e.g., [Sin99]) advocate using statistical modeling and software reliability theory to predict the completeness of testing.

WebRef
A comprehensive glossary of testing terms can be found at **www.testingstandards.co.uk/living_glossary.htm**.

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.

17.2 Strategic Issues

Later in this chapter, I present 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 a software testing strategy will succeed when software testers:

? 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 14). 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 should be stated within the test plan.

WebRef
An excellent list of testing resources can be found at **www.io.com/~wazmo/qa/**.

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 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 antibugging (Section 17.3.1) 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 technical reviews as a filter prior to testing. Technical reviews (Chapter 15) 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 technical reviews to assess the test strategy and test cases themselves. 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.

uote:

"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 Beizer

17.3 Test Strategies for Conventional Software[2]

There are many strategies that can be used to test software. At one extreme, you can 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 all stakeholders. At the other extreme, you 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, some 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.

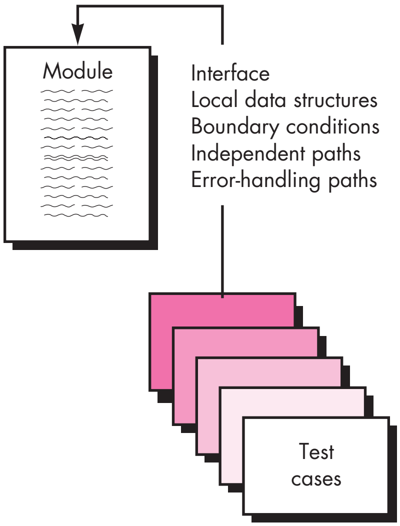
17.3.1 Unit Testing

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

[2] Throughout this book, I use the terms *conventional software* or *traditional software* to refer to common hierarchical or call-and-return software architectures that are frequently encountered in a variety of application domains. Traditional software architectures are *not* object-oriented and do not encompass WebApps.

FIGURE 17.3

Unit test



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.

ADVICE

It's not a bad idea to design unit test cases before *you develop code for a component. It helps ensure that you'll develop code that will pass the tests.*

Unit-test considerations. Unit tests are illustrated schematically in Figure 17.3. 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 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.

Data flow across a component interface is tested before any other testing 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.

? What errors are commonly found 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.

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.

WebRef
Useful information on a wide variety of articles and resources for "agile testing" can be found at **testing.com/agile/**.

A good design anticipates error conditions and establishes error-handling paths 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:

> 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!

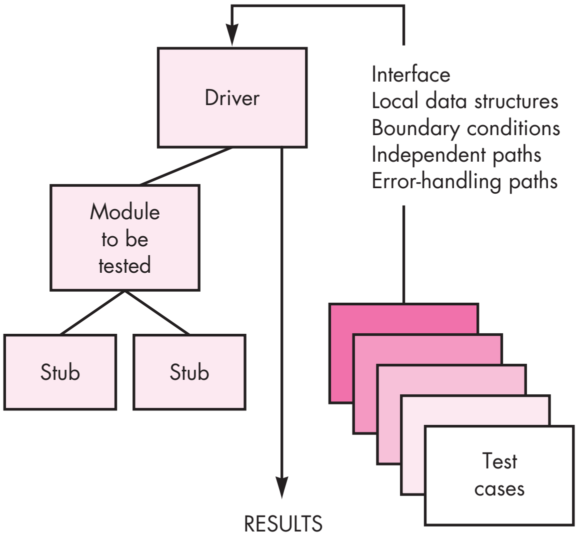
ADVICE
Be sure that you design tests to execute every error-handling path. If you don't, the path may fail when it is invoked, exacerbating an already dicey situation.

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, 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 occur before coding begins 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 often be developed for each unit test. The unit test environment is illustrated in Figure 17.4. 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

FIGURE 17.4
Unit-test environment



relevant results. *Stubs* serve to replace modules that are subordinate (invoked 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 or complex modules and unit test only those.

Drivers and stubs represent testing "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.

17.3.2 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 component can have an inadvertent, adverse effect 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.

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.

ADVICE

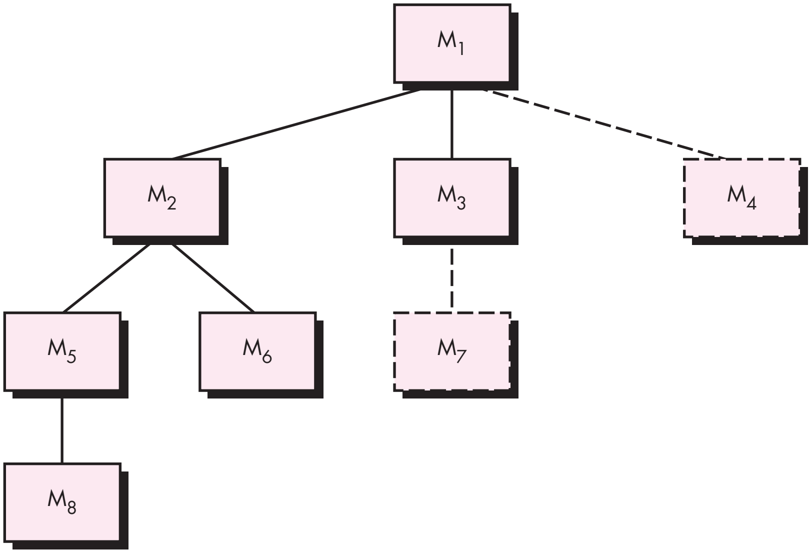
When you develop a project schedule, you'll have to consider the manner in which integration will occur so that components will be available when needed.

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.

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

FIGURE 17.5

Top-down integration



(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 17.5, *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. Early demonstration of functional capability is a confidence builder for all stakeholders.

? 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. As a tester, you are 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) can cause you 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 integration is discussed in the paragraphs that follow.

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, the functionality provided by 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.

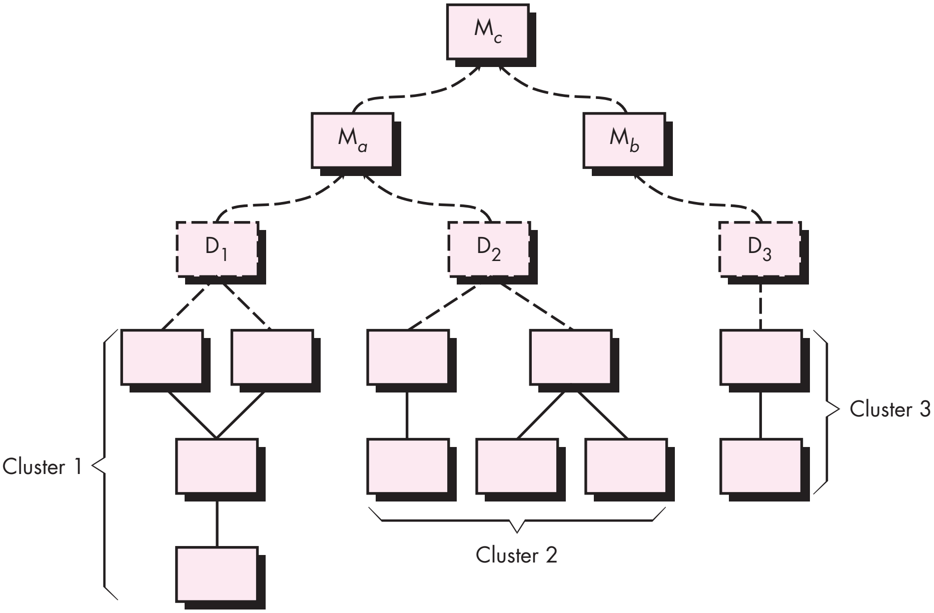
KEY POINT

Bottom-up integration eliminates the need for complex stubs.

Integration follows the pattern illustrated in Figure 17.6. 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.

FIGURE 17.6

Bottom-up integration



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

ADVICE

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

Regression testing. Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the reexecution 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 helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by reexecuting 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 reexecute every test for every program function once a change has occurred.

KEY POINT

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

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

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show-stopper" errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

Quote:

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

Jim McCarthy

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 [McC96] 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 projects:

? What benefits can be derived from smoke testing?

- *Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- *The quality of the end product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.

- *Error diagnosis and correction are simplified.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

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

Strategic options. There has been much discussion (e.g., [Bei84]) about 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.

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

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

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

Integration test work products. An overall plan for integration of the software and a description of specific tests is documented in a *Test Specification.* This work product incorporates a test plan and a test procedure and becomes part of the software configuration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for the *SafeHome* security system might be divided into the following test phases:

- *User interaction* (command input and output, display representation, error processing and representation)
- *Sensor processing* (acquisition of sensor output, determination of sensor conditions, actions required as a consequence of conditions)
- *Communications functions* (ability to communicate with central monitoring station)
- *Alarm processing* (tests of software actions that occur when an alarm is encountered)

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

? What criteria should be used to design integration tests?

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

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

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

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

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

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

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

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

17.4 Test Strategies for Object-Oriented Software[3]

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of object-oriented software changes both testing strategy and testing tactics (Chapter 19).

[3] Basic object-oriented concepts are presented in Appendix 2.

17.4.1 Unit Testing in the OO Context

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

KEY POINT

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

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

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

17.4.2 Integration Testing in the OO Context

Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies (Section 17.3.2) have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class" [Ber93].

KEY POINT

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

There are two different strategies for integration testing of OO systems [Bin94b]. The first, *thread-based testing,* integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

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

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

17.5 Test Strategies for WebApps

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:

KEY POINT
The overall strategy for WebApp testing can be summarized in the 10 steps noted here.

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

WebRef
Excellent articles on WebApp testing can be found at **www.stickyminds.com/testing.asp**.

Because many WebApps evolve continuously, the testing process is an ongoing activity, conducted by support staff who use regression tests derived from the tests developed when the WebApp was first engineered. Methods for WebApp testing are considered in Chapter 20.

17.6 Validation Testing

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

KEY POINT

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

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a *Software Requirements Specification* has been developed, it describes all user-visible attributes of the software and contains a *Validation Criteria* section that forms the basis for a validation-testing approach.

17.6.1 Validation-Test Criteria

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

After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created. Deviations or errors 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.

17.6.2 Configuration Review

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an audit, is discussed in more detail in Chapter 22.

17.6.3 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.

Quote:

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

E. Raymond

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

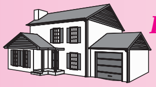
What is the difference between an alpha test and a beta test?

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

The *beta test* is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is 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 tests, you make modifications and then prepare for release of the software product to the entire customer base.

A variation on beta testing, called *customer acceptance testing,* is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SAFEHOME



Preparing for Validation

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

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

The conversation:

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

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

Doug: Talk to me about validation.

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

Ed: Same here.

Jamie: Me too, but we've got to get our act together for acceptance test and also for alpha and beta testing, no?

Doug: Yes. In fact I've been thinking; we could bring in an outside contractor to help us with validation. I have the money in the budget . . . and it'd give us a new point of view.

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

Doug: I'm sure you do, but an ITG gives us an independent look at the software.

Jamie: We're tight on time here, Doug. I for one don't have the time to babysit anybody you bring in to do the job.

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

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

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

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

17.7 System Testing

Quote:

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

Ed Yourdon

At the beginning of this book, I 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 the developers of different system elements blame each other for the problem. Rather than indulging in such nonsense, you should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger pointing does occur, and (4) participate in planning and design of system tests to ensure that software is adequately tested.

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

17.7.1 Recovery Testing

Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

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

17.7.2 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 defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

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

17.7.3 Stress Testing

Earlier software testing steps 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 ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

Quote:

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

Boris Beizer

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

17.7.4 Performance Testing

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

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

17.7.5 Deployment Testing

In many cases, software must execute on a variety of platforms and under more than one operating system environment. *Deployment testing,* sometimes called *configuration testing,* exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

As an example, consider the Internet-accessible version of *SafeHome* software that would allow a customer to monitor the security system from remote locations. The *SafeHome* WebApp must be tested using all Web browsers that are likely to be encountered. A more thorough deployment test might encompass combinations of Web browsers with various operating systems (e.g., Linux, Mac OS, Windows). Because security is a major issue, a complete set of security tests would be integrated with the deployment test.

SOFTWARE TOOLS

Test Planning and Management

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

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

Representative Tools:[4]

QaTraq Test Case Management Tool, developed by Traq Software (**www.testmanagement.com**), "encourages a structured approach to test management."

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

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

OpensourceTesting.org (**www.opensourcetesting.org/testmgt.php**) lists a variety of open-source test management and planning tools.

NI TestStand, developed by National Instruments Corp. (**www.ni.com**), allows you to "develop, manage, and execute test sequences written in any programming language."

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

17.8 The Art of Debugging

Quote:

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

Maurice Wilkes, discovers debugging, 1949

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.

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. As a software engineer, you are often confronted with a "symptomatic" indication of a software problem as you evaluate the results of a test. That is, the external manifestation of the error and its internal cause may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

17.8.1 The Debugging Process

ADVICE

Be certain to avoid a third outcome: The cause is found, but the "correction" does not solve the problem or introduces still another error.

Debugging is not testing but often occurs as a consequence of testing.[5] Referring to Figure 17.7, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

? Why is debugging so difficult?

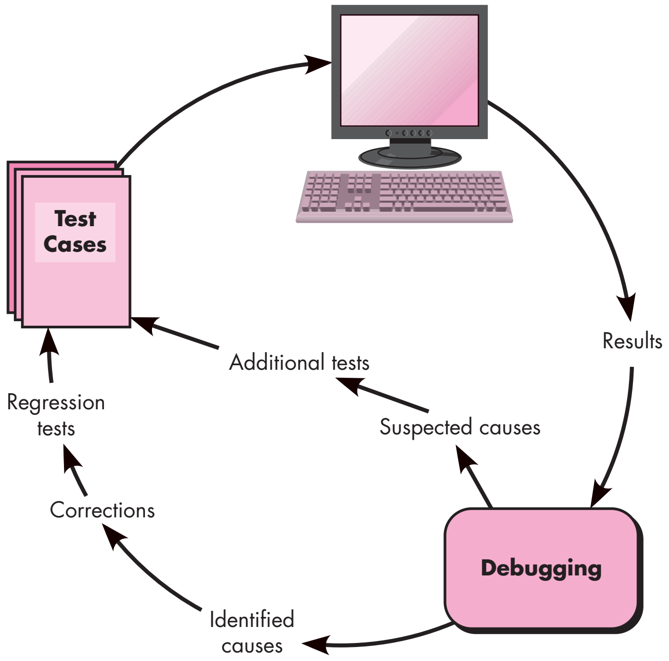
Why is debugging so difficult? In all likelihood, human psychology (see Section 17.8.2) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

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 components (Chapter 8) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.

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

FIGURE 17.7

The debugging process



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

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.

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

17.8.2 Psychological Considerations

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience. Commenting on the human aspects of debugging, Shneiderman [Shn80] states:

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

Although it may be difficult to "learn" debugging, a number of approaches to the problem can be proposed. I examine them in Section 17.8.3.

SAFEHOME

Debugging

The scene: Ed's cubical as code and unit testing is conducted.

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

The conversation:

Shakira (looking in through the entrance to the cubical): Hey . . . where were you at lunchtime?

Ed: Right here . . . working.

Shakira: You look miserable . . . what's the matter?

Ed (sighing audibly): I've been working on this . . . bug since I discovered it at 9:30 this morning and it's what, 2:45 . . . I'm clueless.

Shakira: I thought we all agreed to spend no more than one hour debugging stuff on our own; then we get help, right?

Ed: Yeah, but . . .

Shakira (walking into the cubical): So what's the problem?

Ed: It's complicated, and besides, I've been looking at this for, what, 5 hours. You're not going to see it in 5 minutes.

Shakira: Indulge me . . . what's the problem?

[Ed explains the problem to Shakira, who looks at it for about 30 seconds without speaking, then . . .]

Shakira (a smile is gathering on her face): Uh, right there, the variable named *setAlarmCondition.* Shouldn't it be set to "false" before the loop gets started?

[Ed stares at the screen in disbelief, bends forward, and begins to bang his head gently against the monitor. Shakira, smiling broadly now, stands and walks out.]

17.8.3 Debugging Strategies

Regardless of the approach that is taken, debugging has one overriding objective—to find and correct the cause of a software error or defect. The objective is realized by a combination of systematic evaluation, intuition, and luck. Bradley [Bra85] describes the debugging approach in this way:

ADVICE

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

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

> to see whether the neighborhood is blacked out. I plug the suspect lamp into a working socket and a working appliance into the suspect circuit. So goes the alternation of hypothesis and test.

In general, three debugging strategies have been proposed [Mye79]: (1) brute force, (2) backtracking, and (3) cause elimination. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Quote:

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

T. Duff

Debugging tactics. The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. You hope that somewhere in the morass of information that is produced you'll find a clue that can lead 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 cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

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

Automated debugging. Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted. Hailpern and Santhanam [Hai02] summarize the state of these tools when they note, ". . . many new approaches have been proposed and many commercial debugging environments are available. Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation." A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

SOFTWARE TOOLS

Debugging

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

Mechanics: Most debugging tools are programming language and environment specific.

Representative Tools:[6]

Borland Gauntlet, distributed by Borland (**www.borland.com**), assists in both testing and debugging.

Coverty Prevent SQS, developed by Coverty (**www.coverty.com**), provides debugging assistance for both C++ and Java.

C++Test, developed by Parasoft (**www.parasoft.com**), is a unit-testing tool that supports a full range of tests on C and C++ code. Debugging features assist in the diagnosis of errors that are found.

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

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

The people factor. Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! A fresh viewpoint, unclouded by hours of frustration, can do wonders.[7] A final maxim for debugging might be: "When all else fails, get help!"

17.8.4 Correcting the Error

Quote:

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

Cem Kaner et al.

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 you should ask before making the "correction" that removes the cause of a bug:

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.

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

[7] The concept of pair programming (recommended as part of the Extreme Programming model discussed in Chapter 3) provides a mechanism for "debugging" as the software is designed and coded.

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 16). If you 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.

17.9 Summary

Software testing accounts for the largest percentage of technical effort in the software process. Regardless of the type of software you build, a strategy for systematic test planning, execution, and control begins by considering small elements of the software and moves outward toward the program as a whole.

The objective of software testing is to uncover errors. For conventional software, this objective is achieved through a series of test steps. Unit and integration tests concentrate on functional verification of a component and incorporation of components into the software architecture. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system. Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened.

The strategy for testing object-oriented software begins with tests that exercise the operations within a class and then moves to thread-based testing for integration. Threads are sets of classes that respond to an input or event. Use-based tests focus on classes that do not collaborate heavily with other classes.

WebApps are tested in much the same way as OO systems. However, tests are designed to exercise content, functionality, the interface, navigation, and aspects of WebApp performance and security.

Unlike testing (a systematic, planned activity), debugging can 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.

Problems and Points to Ponder

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

17.2. 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?

17.3. Is it always possible to develop a strategy for testing software that uses the sequence of testing steps described in Section 17.1.3? What possible complications might arise for embedded systems?

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

17.5. The concept of "antibugging" (Section 17.2.1) is an extremely effective way to provide built-in debugging assistance when an error is uncovered:

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

17.6. How can project scheduling affect integration testing?

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

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

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

17.10. 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 Sources

Virtually every book on software testing discusses strategies along with methods for test-case design. Everett and Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller and his colleagues (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2d ed., Auerbach, 2004), Loveland and his colleagues (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig and Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), Whittaker (*How to Break Software,* Addison-Wesley, 2002), and Kaner and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001) are only a small sampling of many books that discuss testing principles, concepts, strategies, and methods.

For those readers with interest in agile software development methods, Crispin and House (*Testing Extreme Programming,* Addison-Wesley, 2002) and Beck (*Test Driven Development: By Example,* Addison-Wesley, 2002) present testing strategies and tactics for Extreme Programming. Kamer and his colleagues (*Lessons Learned in Software Testing,* Wiley, 2001) present a collection of over 300 pragmatic "lessons" (guidelines) that every software tester should learn. Watkins (*Testing IT: An Off-the-Shelf Testing Process,* Cambridge University Press, 2001) establishes an effective testing framework for all types of developed and acquired software. Manges and O'Brien (*Agile Testing with Ruby and Rails,* Apress, 2008) addresses testing strategies and techniques for the Ruby programming language and Web framework.

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

Guidelines for debugging are contained in books by Grötker and his colleagues (*The Developer's Guide to Debugging,* Springer, 2008), Agans (*Debugging,* Amacon, 2006), Zeller (*Why Programs Fail: A Guide to Systematic Debugging,* Morgan Kaufmann, 2005), Tells and Hsieh (*The Science of Debugging,* The Coreolis Group, 2001), and Robbins (*Debugging Applications,* Microsoft Press, 2000). Kaspersky (*Hacker Debugging Uncovered,* A-List Publishing, 2005) addresses the technology of debugging tools. Younessi (*Object-Oriented Defect Management of Software,* Prentice-Hall, 2002) presents techniques for managing defects that are encountered in object-oriented systems. Beizer [Bei84] presents an interesting "taxonomy of bugs" that can lead to effective methods for test planning.

Books by Madisetti and Akgul (*Debugging Embedded Systems,* Springer, 2007), Robbins (*Debugging Microsoft .NET 2.0 Applications,* Microsoft Press, 2005), Best (*Linux Debugging and Performance Tuning,* Prentice-Hall, 2005), Ford and Teorey (*Practical Debugging in C++,* Prentice-Hall, 2002), Brown (*Debugging Perl,* McGraw-Hill, 2000), and Mitchell (*Debugging Java,* McGraw-Hill, 2000) address the special nature of debugging for the environments implied by their titles.

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

CHAPTER

Testing Conventional Applications 18

Key Concepts

Testing presents an interesting anomaly for software engineers, who by their nature are constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and then work hard to design test cases to "break" the software. Beizer [Bei90] describes this situation effectively when he states:

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

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

In this chapter, I discuss techniques for software test-case design for conventional applications. Test-case design focuses on a set of techniques for the creation of test cases that meet overall testing objectives and the testing strategies discussed in Chapter 17.

Quick Look

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

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

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

What are the steps? For conventional applications, software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test-case design techniques and (2) software requirements are exercised using "black box" test-case design techniques. Use cases assist in the design of tests to uncover errors at the software validation level. In every case, the intent is to find the maximum number of errors with the minimum amount of effort and time.

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

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

18.1 Software Testing Fundamentals

Quote:

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

Author unknown

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

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

? What are the characteristics of testability?

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

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

Controllability. "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and

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

variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.

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

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

uote:

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

David Parnas

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

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

You can use the attributes suggested by Bach to develop a software configuration (i.e., programs, data, and documents) that is amenable to testing.

Test Characteristics. And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a "good" test:

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

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

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

A good test should be "best of breed" [Kan93]. In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

SafeHome

Designing Unique Tests

The scene: Vinod's cubical.

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

The conversation:

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

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

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

Ed: Uh huh.

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

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

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

Ed: Well, they're different values.

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

Ed: I see what you mean.

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

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

18.2 Internal and External Views of Testing

Quote:

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

Tsuneo Yamaura

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

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

KEY POINT

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

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents

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

certain logistical problems. For even small programs, the number of possible logical paths can be very large. White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

INFO

Exhaustive Testing

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

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

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

18.3 White-Box Testing

Quote:

"Bugs lurk in corners and congregate at boundaries."

Boris Beizer

White-box testing, sometimes called *glass-box testing,* is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

18.4 Basis Path Testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe [McC76]. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

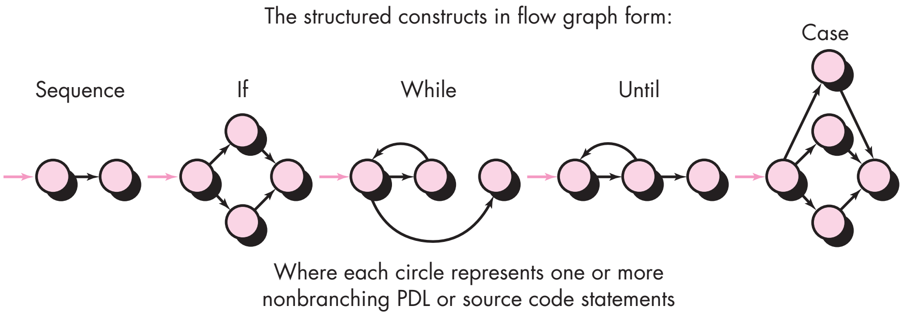
18.4.1 Flow Graph Notation

Before we consider the basis path method, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced.[3] The flow graph depicts logical control flow using the notation illustrated in Figure 18.1. Each structured construct (Chapter 10) has a corresponding flow graph symbol.

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

FIGURE 18.1

Flow graph notation

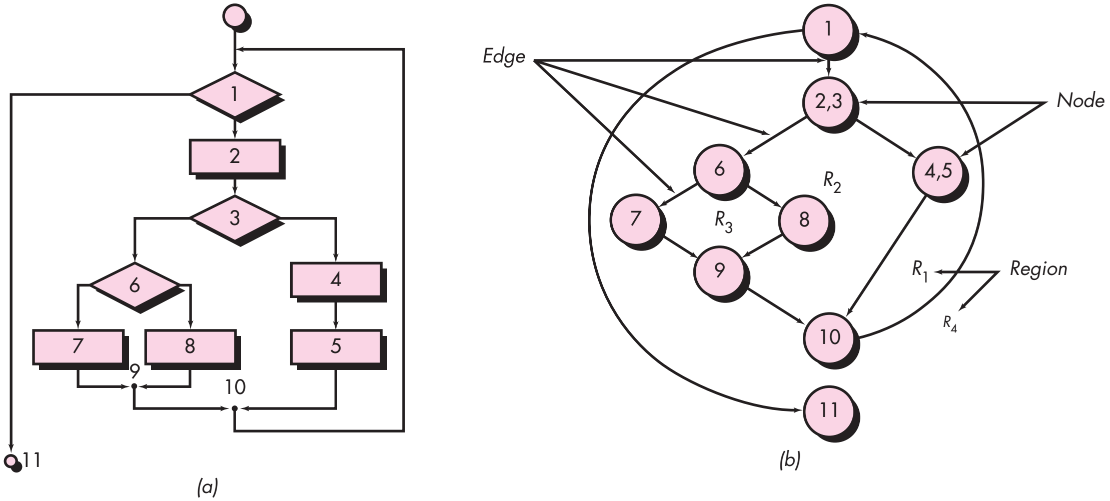

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

ADVICE

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

To illustrate the use of a flow graph, consider the procedural design representation in Figure 18.2a. Here, a flowchart is used to depict program control structure. Figure 18.2b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 18.2b, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links,* represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region.[4]

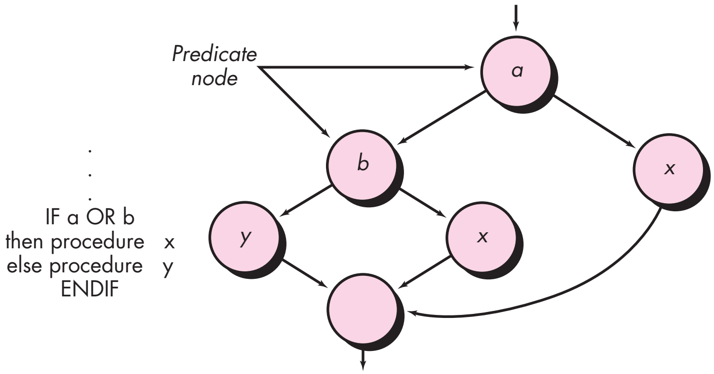
FIGURE 18.2 (a) Flowchart and (b) flow graph



[4] A more detailed discussion of graphs and their uses is presented in Section 18.6.1.

FIGURE 18.3

Compound logic



When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 18.3, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b.* Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

18.4.2 Independent Program Paths

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 18.2b is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

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

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

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

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

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

Paths 1 through 4 constitute a *basis set* for the flow graph in Figure 18.2b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

ADVICE

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

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

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

? How do I compute cyclomatic complexity?

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

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

   where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

   $V(G) = P + 1$

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

Referring once more to the flow graph in Figure 18.2b, the cyclomatic complexity can be computed using each of the algorithms just noted:

KEY POINT

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

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

Therefore, the cyclomatic complexity of the flow graph in Figure 18.2b is 4.

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

SAFEHOME

Using Cyclomatic Complexity

The scene: Shakira's cubicle.

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

The conversation:

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

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

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

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

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

Vinod: *V* of *G*.

Shakira: Huh?

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

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

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

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

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

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

18.4.3 Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, I present basis path testing as a series of steps. The procedure *average,* depicted in PDL in Figure 18.4, will be used as an example to illustrate each step in the test-case design method. Note that *average,* although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

Quote:

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

A news report

1. **Using the design or code as a foundation, draw a corresponding flow graph.** A flow graph is created using the symbols and construction rules presented in Section 18.4.1. Referring to the PDL for *average* in Figure 18.4, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is shown in Figure 18.5.
2. **Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity *V*(*G*) is determined by applying the algorithms described in Section 18.4.2. It should be noted that *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1. Referring to Figure 18.5,

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

FIGURE 18.4

PDL with nodes identified

```
PROCEDURE average;

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

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

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

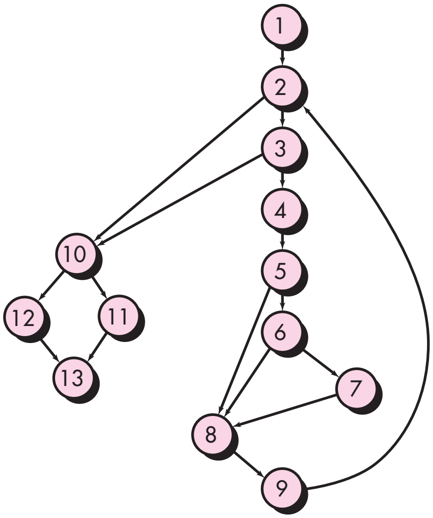
3. **Determine a basis set of linearly independent paths.** The value of $V(G)$ provides the upper bound on the number of linearly independent paths through the program control structure. In the case of procedure *average,* we expect to specify six paths:

Path 1: 1-2-10-11-13

Path 2: 1-2-10-12-13

FIGURE 18.5

Flow graph for the procedure *average*



Path 3: 1-2-3-10-11-13

Path 4: 1-2-3-4-5-8-9-2-. . .

Path 5: 1-2-3-4-5-6-8-9-2-. . .

Path 6: 1-2-3-4-5-6-7-8-9-2-. . .

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

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

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

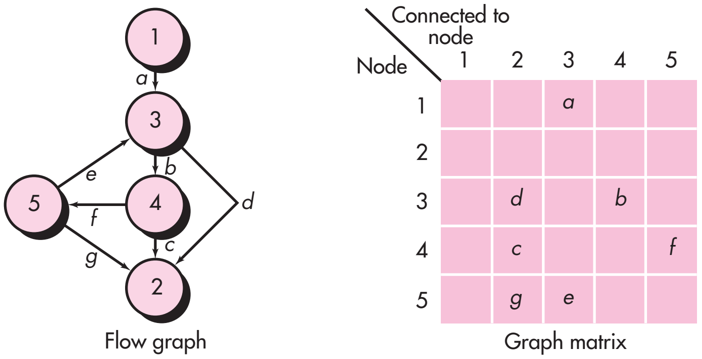
18.4.4 Graph Matrices

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

A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix [Bei90] is shown in Figure 18.6.

FIGURE 18.6

Graph matrix



Node \ Connected to node	1	2	3	4	5
1			*a*		
2					
3		*d*		*b*	
4		*c*			*f*
5		*g*	*e*		

Graph matrix

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

? What is a graph matrix and how do I extend it for use in testing?

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

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

Beizer [Bei90] provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

18.5 Control Structure Testing

Quote:

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

Brian Marick

The basis path testing technique described in Section 18.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

18.5.1 Condition Testing

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E_1 <relational-operator> E_2

KEY POINT

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

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

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

uote:

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

Brian Marick

18.5.2 Data Flow Testing

The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

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

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

ADVICE

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

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

One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the *then part* has no definition of any variable and the *else part* does not exist. In this situation, the else branch of the *if* statement is not necessarily covered by DU testing.

18.5.3 Loop Testing

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

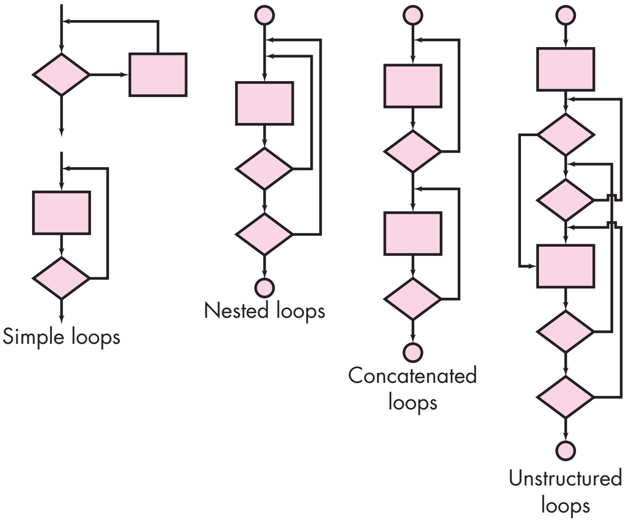
Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [Bei90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 18.7).

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

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.

FIGURE 18.7

Classes of Loops



4. m passes through the loop where $m < n$.
5. $n - 1, n, n + 1$ passes through the loop.

Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

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

ADVICE

You can't test unstructured loops effectively. Refactor them.

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

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 10).

18.6 Black-Box Testing

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

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

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing (see Chapter 17). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

Quote:

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

Robert Dunn

What questions do black-box tests answer?

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

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria [Mye79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

KEY POINT

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

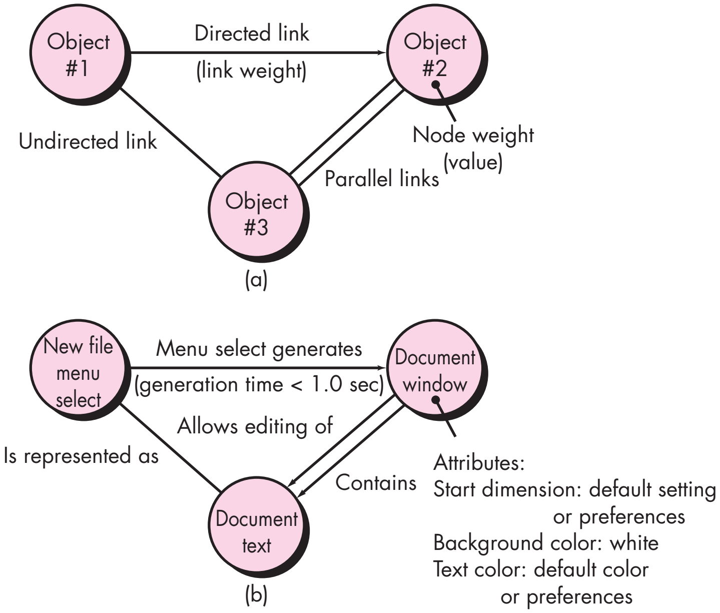
18.6.1 Graph-Based Testing Methods

The first step in black-box testing is to understand the objects[5] that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another" [Bei95]. Stated in another way, software testing begins by creating a graph of important objects and their relationships and

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

FIGURE 18.8

(a) Graph notation; (b) simple example



then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

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

The symbolic representation of a graph is shown in Figure 18.8a. Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 18.8b) where

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

Object #2 = **documentWindow**

Object #3 = **documentText**

Referring to the figure, a menu select on **newFile** generates a document window. The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.** In reality, a far more detailed graph would have to be generated as a precursor to test-case design. You can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer [Bei95] describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps (e.g., **flightInformationInput** is followed by *validationAvailabilityProcessing*). The data flow diagram (Chapter 7) can be used to assist in creating graphs of this type.

Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **orderInformation** is verified during *inventoryAvailabilityLook-up* and is followed by **customerBillingInformation** input). The state diagram (Chapter 7) can be used to assist in creating graphs of this type.

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

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

A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. If you have further interest, see [Bei95] for a comprehensive coverage.

18.6.2 Equivalence Partitioning

ADVICE

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

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

Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [Bei95]. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

? How do I define equivalence classes for testing?

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

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

18.6.3 Boundary Value Analysis

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

Quote:

"An effective way to test code is to exercise it at its natural boundaries."

Brian Kernighan

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [Mye79].

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

KEY POINT

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

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

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

18.6.4 Orthogonal Array Testing

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

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

KEY POINT

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

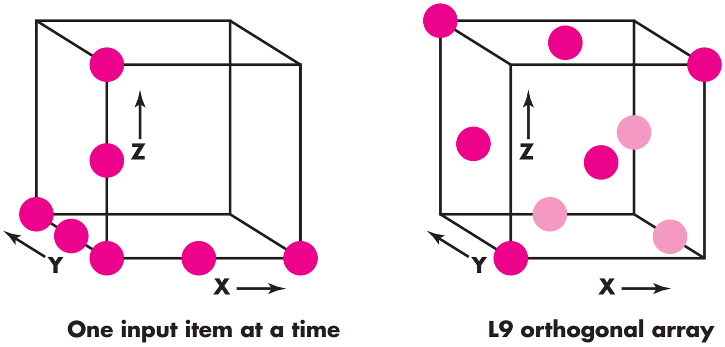
To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, *X, Y,* and *Z.* Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases. Phadke [Pha97] suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 18.9. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

When orthogonal array testing occurs, an L9 *orthogonal array* of test cases is created. The L9 orthogonal array has a "balancing property" [Pha97]. That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure 18.9. Test coverage across the input domain is more complete.

FIGURE 18.9

A geometric view of test cases

Source: [Pha97]



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

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

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

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). Phadke [Pha97] assesses these test cases by stating:

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

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

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax *send* function is illustrated in Figure 18.10.

FIGURE 18.10

An L9 orthogonal array

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Phadke [Pha97] assesses the result of tests using the L9 orthogonal array in the following manner:

> **Detect and isolate all single mode faults.** A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 18.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.
>
> **Detect all double mode faults.** If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault.* Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.
>
> **Multimode faults.** Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multimode faults are also detected by these tests.

You can find a detailed discussion of orthogonal array testing in [Pha89].

SOFTWARE TOOLS

Test-Case Design

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

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

Representative Tools:[6]

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

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

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

e-TEST Suite, developed by Empirix, Inc. (**www.empirix.com**), encompasses a complete set of tools for testing WebApps, including tools that assist test-case design and test planning.

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

18.7 Model-Based Testing

> **Quote:**
>
> "It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free."
>
> **Steve McConnell**

Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases. In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model (Chapter 7), as the basis for the design of test cases.[7] The MBT technique requires five steps:

1. **Analyze an existing behavioral model for the software or create one.** Recall that a *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the steps discussed in Chapter 7: (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a UML state diagram for the system (e.g., see Figure 7.6), and (5) review the behavioral model to verify accuracy and consistency.
2. **Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.** The inputs will trigger events that will cause the transition to occur.
3. **Review the behavioral model and note the expected outputs as the software makes the transition from state to state.** Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created. For each set of inputs (test cases) you specified in step 2, specify the expected outputs as they are characterized in the behavioral model. "A fundamental assumption of this testing is that there is some mechanism, a *test oracle*, that will determine whether or not the results of a test execution are correct" [DAC03]. In essence, a test oracle establishes the basis for any determination of the correctness of the output. In most cases, the oracle is the requirements model, but it could also be another document or application, data recorded elsewhere, or even a human expert.
4. **Execute the test cases.** Tests can be executed manually or a test script can be created and executed using a testing tool.
5. **Compare actual and expected results and take corrective action as required.**

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

[7] Model-based testing can also be used when software requirements are represented with decision tables, grammars, or Markov chains [DAC03].

18.8 Testing for Specialized Environments, Architectures, and Applications

Unique guidelines and approaches to testing are sometimes warranted when specialized environments, architectures, and applications are considered. Although the testing techniques discussed earlier in this chapter and in Chapters 19 and 20 can often be adapted to specialized situations, it's worth considering their unique needs individually.

18.8.1 Testing GUIs

Graphical user interfaces (GUIs) will present you with interesting testing challenges. Because reusable components are now a common part of GUI development environments, the creation of the user interface has become less time consuming and more precise (Chapter 11). But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite-state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI. This model-based testing technique was discussed in Section 18.7.

Because of the large number of permutations associated with GUI operations, GUI testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.[8]

Quote:

"The topic of testing is one area in which a good deal of commonality exists between traditional system and client/server systems."

Kelley Bourne

WebRef

Useful client-sever testing information and resources can be found at **www.csst-technologies.com**.

18.8.2 Testing of Client-Server Architectures

The distributed nature of client-server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of client-server architectures and the software that resides within them considerably more difficult than stand-alone applications. In fact, recent industry studies indicate a significant increase in testing time and cost when client-server environments are developed.

In general, the testing of client-server software occurs at three different levels: (1) Individual client applications are tested in a "disconnected" mode; the operation of the server and the underlying network are not considered. (2) The client software and associated server applications are tested in concert, but network operations are not explicitly exercised. (3) The complete client-server architecture, including network operation and performance, is tested.

[8] Hundreds, if not thousands, of GUI testing tool resources can be evaluated on the Web. A good starting point for open-source tools is **www.opensourcetesting.org/functional.php**.

Although many different types of tests are conducted at each of these levels of detail, the following testing approaches are commonly encountered for client-server applications:

? What types of tests are conducted for client-server systems?

- **Application function tests.** The functionality of client applications is tested using the methods discussed earlier in this chapter and in Chapters 19 and 20. In essence, the application is tested in stand-alone fashion in an attempt to uncover errors in its operation.
- **Server tests.** The coordination and data management functions of the server are tested. Server performance (overall response time and data throughput) is also considered.
- **Database tests.** The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated, and retrieved. Archiving is also tested.
- **Transaction tests.** A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues (e.g., transaction processing times and transaction volume).
- **Network communication tests.** These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests.

To accomplish these testing approaches, Musa [Mus93] recommends the development of *operational profiles* derived from client-server usage scenarios.[9] An operational profile indicates how different types of users interoperate with the client-server system. That is, the profiles provide a "pattern of usage" that can be applied when tests are designed and executed. For example, for a particular type of user, what percentage of transactions will be inquiries? updates? orders?

To develop the operational profile, it is necessary to derive a set of scenarios that are similar to use cases (Chapters 5 and 6). Each scenario addresses who, where, what, and why. That is, who the user is, where (in the physical client-server architecture) the system interaction occurs, what the transaction is, and why it has occurred. Scenarios can be derived using requirements elicitation techniques (Chapter 5) or through less formal discussions with end users. The result, however, should be the same. Each scenario should provide an indication of the system functions that will be required to service a particular user, the order in which those functions are required, the timing and response that is expected, and the frequency with which each function is used. These data are then combined (for all users) to create the operational profile. In general, testing effort and the number of test cases to be executed are

[9] It should be noted that operational profiles can be used in testing for all types of system architectures, not just client-server architecture.

allocated to each usage scenario based on frequency of usage and criticality of the functions performed.

18.8.3 Testing Documentation and Help Facilities

The term *software testing* conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. Recalling the definition of software presented in Chapter 1, it is important to note that testing must also extend to the third element of the software configuration—documentation.

Errors in documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more frustrating than following a user guide or an online help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. It is for this reason that that documentation testing should be a meaningful part of every software test plan.

Documentation testing can be approached in two phases. The first phase, technical review (Chapter 15), examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. MBT can be used to ensure that documented behavior and actual behavior coincide. Program usage is then tracked through the documentation.

INFO

Documentation Testing

The following questions should be answered during documentation and/or help facility testing:

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are the document's table of contents and index robust, accurate, and complete?
- Is the design of the document (layout, typefaces, indentation, graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted and areas of document ambiguity or weakness are defined for potential rewrite.

18.8.4 Testing for Real-Time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix—time. Not only does the test-case designer have to consider conventional test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. In many situations, test data provided when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.

For example, the real-time software that controls a new photocopier accepts operator interrupts (i.e., the machine operator hits control keys such as RESET or DARKEN) with no error when the machine is making copies (in the "copying" state). These same operator interrupts, if input when the machine is in the "jammed" state, cause a display of the diagnostic code indicating the location of the jam to be lost (an error).

In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.

Comprehensive test-case design methods for real-time systems continue to evolve. However, an overall four-step strategy can be proposed:

? What is an effective strategy for testing a real-time system?

- **Task testing.** The first step in the testing of real-time software is to test each task independently. That is, conventional tests are designed for each task and executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior.
- **Behavioral testing.** Using system models created with automated tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built. Using a technique that is similar to equivalence partitioning (Section 18.6.2), events (e.g., interrupts, control signals) are categorized for testing. For example, events for the photocopier might be user interrupts (e.g., reset counter), mechanical interrupts (e.g., paper jammed), system interrupts (e.g., toner low), and failure modes (e.g., roller overheated). Each of these events is tested individually, and the behavior of the executable system is examined to detect errors that occur as a consequence of processing associated with these events. The behavior of the system model (developed during the analysis activity) and the executable software can be compared for conformance. Once each class of events has been tested, events are presented to the system in random order and with random frequency. The behavior of the software is examined to detect behavior errors.

- **Intertask testing.** Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
- **System testing.** Software and hardware are integrated, and a full range of system tests are conducted in an attempt to uncover errors at the software-hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential. Using the state diagram (Chapter 7), the tester develops a list of all possible interrupts and the processing that occurs as a consequence of the interrupts. Tests are then designed to assess the following system characteristics:
    - Are interrupt priorities properly assigned and properly handled?
    - Is processing for each interrupt handled correctly?
    - Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?
    - Does a high volume of interrupts arriving at critical times create problems in function or performance?

In addition, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

18.9 Patterns for Software Testing

The use of patterns as a mechanism for describing solutions to specific design problems was discussed in Chapter 12. But patterns can also be used to propose solutions to other software engineering situations—in this case, software testing. *Testing patterns* describe common testing problems and solutions that can assist you in dealing with them.

WebRef

A software testing patterns catalog can be found at **www.rbsc.com/pages/TestPatternList.htm**.

Not only do testing patterns provide you with useful guidance as testing activities commence, they also provide three additional benefits described by Marick [Mar02]:

1. They [patterns] provide a vocabulary for problem-solvers. "Hey, you know, we should use a Null Object."
2. They focus attention on the forces behind a problem. That allows [test case] designers to better understand when and why a solution applies.
3. They encourage iterative thinking. Each solution creates a new context in which new problems can be solved.

KEY POINT

Testing patterns can help a software team communicate more effectively about testing and better understand the forces that lead to a specific testing approach.

Although these benefits are "soft," they should not be overlooked. Much of software testing, even during the past decade, has been an ad hoc activity. If testing patterns can help a software team to communicate about testing more effectively;

WebRef

Patterns that describe testing organization, efficiency, strategy, and problem resolution can be found at: **www.testing.com/test-patterns/patterns/**.

to understand the motivating forces that lead to a specific approach to testing, and to approach the design of tests as an evolutionary activity in which each iteration results in a more complete suite of test cases, then patterns have accomplished much.

Testing patterns are described in much the same way as design patterns (Chapter 12). Dozens of testing patterns have been proposed in the literature (e.g., [Mar02]). The following three testing patterns (presented in abstract form only) provide representative examples:

Pattern name: **PairTesting**

Abstract: A process-oriented pattern, **PairTesting** describes a technique that is analogous to pair programming (Chapter 3) in which two testers work together to design and execute a series of tests that can be applied to unit, integration or validation testing activities.

Pattern name: **SeparateTestInterface**

Abstract: There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The **SeparateTestInterface** pattern describes how to create "a test interface that can be used to describe specific tests on classes that are visible only internally to a component" [Lan01].

Pattern name: **ScenarioTesting**

Abstract: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement [Kan01].

A comprehensive discussion of testing patterns is beyond the scope of this book. If you have further interest, see [Bin99] and [Mar02] for additional information on this important topic.

18.10 Summary

The primary objective for test-case design is to derive a set of tests that have the highest likelihood for uncovering errors in software. To accomplish this objective, two different categories of test-case design techniques are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure statement coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.

Hetzel [Het84] describes white-box testing as "testing in the small." His implication is that the white-box tests that have been considered in this chapter are typically applied to small program components (e.g., modules or small groups of modules). Black-box testing, on the other hand, broadens your focus and might be called "testing in the large."

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters. Model-based testing uses elements of the requirements model to test the behavior of an application.

Specialized testing methods encompass a broad array of software capabilities and application areas. Testing for graphical user interfaces, client-server architectures, documentation and help facilities, and real-time systems each require specialized guidelines and techniques.

Experienced software developers often say, "Testing never ends, it just gets transferred from you [the software engineer] to your customer. Every time your customer uses the program, a test is being conducted." By applying test-case design, you can achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

Problems and Points to Ponder

18.1. Myers [Mye79] uses the following program as a self-assessment for your ability to specify adequate testing: A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.

18.2. Design and implement the program (with error handling where appropriate) specified in Problem 18.1. Derive a flow graph for the program and apply basis path testing to develop test cases that will guarantee that all statements in the program have been tested. Execute the cases and show your results.

18.3. Can you think of any additional testing objectives that are not discussed in Section 18.1.1?

18.4. Select a software component that you have designed and implemented recently. Design a set of test cases that will ensure that all statements have been executed using basis path testing.

18.5. Specify, design, and implement a software tool that will compute the cyclomatic complexity for the programming language of your choice. Use the graph matrix as the operative data structure in your design.

18.6. Read Beizer [Bei95] or a related Web-based source (e.g., **www.laynetworks.com/Discrete%20Mathematics_1g.htm**) and determine how the program you have developed in Problem 18.5 can be extended to accommodate various link weights. Extend your tool to process execution probabilities or link processing times.

18.7. Design an automated tool that will recognize loops and categorize them as indicated in Section 18.5.3.

18.8. Extend the tool described in Problem 18.7 to generate test cases for each loop category, once encountered. It will be necessary to perform this function interactively with the tester.

18.9. Give at least three examples in which black-box testing might give the impression that "everything's OK," while white-box tests might uncover an error. Give at least three examples in which white-box testing might give the impression that "everything's OK," while black-box tests might uncover an error.

18.10. Will exhaustive testing (even if it is possible for very small programs) guarantee that the program is 100 percent correct?

18.11. Test a user manual (or help facility) for an application that you use frequently. Find at least one error in the documentation.

Further Readings and Information Sources

Virtually all books dedicated to software testing consider both strategy and tactics. Therefore, further readings noted for Chapter 17 are equally applicable for this chapter. Everett and Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller and his colleagues (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2d ed., Auerbach, 2004), Loveland and his colleagues (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig and Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), and Whittaker (*How to Break Software,* Addison-Wesley, 2002) are only a small sampling of many books that discuss testing principles, concepts, strategies, and methods.

A second edition of Myers [Mye79] classic text has been produced by Myers and his colleagues (*The Art of Software Testing,* 2d ed., Wiley, 2004) and covers test-case design techniques in considerable detail. Pezze and Young (*Software Testing and Analysis,* Wiley, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design,* Artech, 2003), Hutcheson (*Software Testing Fundamentals,* Wiley, 2003), Jorgensen (*Software Testing: A Craftsman's Approach,* 2d ed., CRC Press, 2002) each provide useful presentations of test-case design methods and techniques. Beizer's [Bei90] classic text provides comprehensive coverage of white-box techniques, introducing a level of mathematical rigor that has often been missing in other treatments of testing. His later book [Bei95] presents a concise treatment of important methods.

Software testing is a resource-intensive activity. It is for this reason that many organizations automate parts of the testing process. Books by Li and Wu (*Effective Software Test Automation,* Sybex, 2004); Mosely and Posey (*Just Enough Software Test Automation,* Prentice-Hall, 2002); Dustin, Rashka, and Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999); Graham and her colleagues (*Software Test Automation*, Addison-Wesley, 1999); and Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) discuss tools, strategies, and methods for automated testing. Nquyen and his colleagues (*Global Software Test Automation,* Happy About Press, 2006) present an executive overview of testing automation.

Thomas and his colleagues (*Java Testing Patterns,* Wiley, 2004) and Binder [Bin99] describe testing patterns that cover testing of methods, classes/clusters, subsystems, reusable components, frameworks, and systems as well as test automation and specialized database testing.

A wide variety of information sources on test-case design methods is available on the Internet. An up-to-date list of World Wide Web references that are relevant to testing techniques can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PART

Four

Managing Software Projects

In this part of *Software Engineering: A Practitioner's Approach* you'll learn the management techniques required to plan, organize, monitor, and control software projects. These questions are addressed in the chapters that follow:

- How must people, process, and problem be managed during a software project?
- How can software metrics be used to manage a software project and the software process?
- How does a software team generate reliable estimates of effort, cost, and project duration?
- What techniques can be used to assess the risks that can have an impact on project success?
- How does a software project manager select a set of software engineering work tasks?
- How is a project schedule created?
- Why are maintenance and reengineering so important for both software engineering managers and practitioners?

Once these questions are answered, you'll be better prepared to manage software projects in a way that will lead to timely delivery of a high-quality product.

CHAPTER

24 Project Management Concepts

In the preface to his book on software project management, Meiler Page-Jones [Pag85] makes a statement that can be echoed by many software engineering consultants:

> I've visited dozens of commercial shops, both good and bad, and I've observed scores of data processing managers, again, both good and bad. Too often, I've watched in horror as these managers futilely struggled through nightmarish projects, squirmed under impossible deadlines, or delivered systems that outraged their users and went on to devour huge chunks of maintenance time.

What Page-Jones describes are symptoms that result from an array of management and technical problems. However, if a post mortem were to be conducted

Quick Look

What is it? Although many of us (in our darker moments) take Dilbert's view of "management," it remains a very necessary activity when computer-based systems and products are built. Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to full operational deployment.

Who does it? Everyone "manages" to some extent, but the scope of management activities varies among people involved in a software project. A software engineer manages her day-to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and software professionals.

Why is it important? Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed.

What are the steps? Understand the four P's—people, product, process, and project. People must be organized to perform software work effectively. Communication with the customer and other stakeholders must occur so that product scope and requirements are understood. A process that is appropriate for the people and the product should be selected. The project must be planned by estimating effort and calendar time to accomplish work tasks: defining work products, establishing quality checkpoints, and identifying mechanisms to monitor and control work defined by the plan.

What is the work product? A project plan is produced as management activities commence. The plan defines the process and tasks to be conducted, the people who will do the work, and the mechanisms for assessing risks, controlling change, and evaluating quality.

How do I ensure that I've done it right? You're never completely sure that the project plan is right until you've delivered a high-quality product on time and within budget. However, a project manager does it right when he encourages software people to work together as an effective team, focusing their attention on customer needs and product quality.

for every project, it is very likely that a consistent theme would be encountered: project management was weak.

In this chapter and Chapters 25 through 29, I'll present the key concepts that lead to effective software project management. This chapter considers basic software project management concepts and principles. Chapter 25 presents process and project metrics, the basis for effective management decision making. The techniques that are used to estimate cost are discussed in Chapter 26. Chapter 27 will help you to define a realistic project schedule. The management activities that lead to effective risk monitoring, mitigation, and management are presented in Chapter 28. Finally, Chapter 29 considers maintenance and reengineering and discusses the management issues that you'll encounter when you must deal with legacy systems.

24.1 The Management Spectrum

Effective software project management focuses on the four P's: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

24.1.1 The People

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a *People Capability Maturity Model* (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives" [Cur01].

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices.

The People-CMM is a companion to the *Software Capability Maturity Model–Integration* (Chapter 30) that guides organizations in the creation of a mature software process. Issues associated with people management and structure for software projects are considered later in this chapter.

24.1.2 The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering (Chapter 5). Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

24.1.3 The Process

A software process (Chapters 2 and 3) provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

ADVICE

Those who adhere to the agile process philosophy (Chapter 3) argue that their process is leaner than others. That may be true, but they still have a process, and agile software engineering still requires discipline.

24.1.4 The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones [Jon04] found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below

35 percent, while about 175 experienced major delays and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be.[1]

To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 24.5 and in the chapters that follow.

24.2 People

In a study published by the IEEE [Cur88], the engineering vice presidents of three major technology companies were asked what was the most important contributor to a successful software project. They answered in the following way:

> **VP 1:** I guess if you had to pick one thing out that is most important in our environment, I'd say it's not the tools that we use, it's the people.
>
> **VP 2:** The most important ingredient that was successful on this project was having smart people . . . very little else matters in my opinion. . . . The most important thing you do for a project is selecting the staff. . . . The success of the software development organization is very, very much associated with the ability to recruit good people.
>
> **VP 3:** The only rule I have in management is to ensure I have good people—real good people—and that I grow good people—and that I provide an environment in which good people can produce.

Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue (as the preceding group had) that people are primary, but their actions sometimes belie their words. In this section I examine the stakeholders who participate in the software process and the manner in which they are organized to perform effective software engineering.

24.2.1 The Stakeholders

The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

1. *Senior managers* who define the business issues that often have a significant influence on the project.

[1] Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially. Part of the answer, I think, is that a substantial number of these "failed" projects are ill conceived in the first place. Customers lose interest quickly (because what they've requested wasn't really as important as they first thought), and the projects are cancelled.

2. *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
3. *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.
4. *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
5. *End users* who interact with the software once it is released for production use.

Every software project is populated by people who fall within this taxonomy.[2] To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. And that's the job of the team leader.

24.2.2 Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders. They simply don't have the right mix of people skills. And yet, as Edgemon states: "Unfortunately and all too frequently it seems, individuals just fall into a project manager role and become accidental project managers" [Edg95].

In an excellent book of technical leadership, Jerry Weinberg [Wei86] suggests an MOI model of leadership:

? What do we look for when choosing someone to lead a software project?

Motivation. The ability to encourage (by "push or pull") technical people to produce to their best ability.

Organization. The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem-solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know (by words and, far more important, by actions) that quality counts and that it will not be compromised.

Quote:

"In simplest terms, a leader is one who knows where he wants to go, and gets up, and goes."

John Erskine

Another view [Edg95] of the characteristics that define an effective project manager emphasizes four key traits:

Problem solving. An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and

[2] When WebApps are developed, other nontechnical people may be involved in content creation.

remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity. A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement. A competent manager must reward initiative and accomplishment to optimize the productivity of a project team. She must demonstrate through her own actions that controlled risk taking will not be punished.

Influence and team building. An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

24.2.3 The Software Team

Quote:

"Not every group is a team, and not every team is effective."

Glenn Parker

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change are not within the software project manager's scope of responsibility. However, the organization of the people directly involved in a new software project is within the project manager's purview.

The "best" team structure depends on the management style of your organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty. Mantei [Man81] describes seven project factors that should be considered when planning the structure of software engineering teams:

? What factors should be considered when the structure of a software team is chosen?

- Difficulty of the problem to be solved
- "Size" of the resultant program(s) in lines of code or function points
- Time that the team will stay together (team lifetime)
- Degree to which the problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date
- Degree of sociability (communication) required for the project

Constantine [Con93] suggests four "organizational paradigms" for software engineering teams:

? What options do we have when defining the structure of a software team?

1. A *closed paradigm* structures a team along a traditional hierarchy of authority. Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.

2. A *random paradigm* structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may struggle when "orderly performance" is required.
3. An *open paradigm* attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively, with heavy communication and consensus-based decision making the trademarks of open paradigm teams. Open paradigm team structures are well suited to the solution of complex problems but may not perform as efficiently as other teams.
4. A *synchronous paradigm* relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves.

Quote:

"If you want to be incrementally better: Be competitive. If you want to be exponentially better: Be cooperative."

Author unknown

As an historical footnote, one of the earliest software team organizations was a closed paradigm structure originally called the *chief programmer team.* This structure was first proposed by Harlan Mills and described by Baker [Bak72]. The nucleus of the team was composed of a *senior engineer* (the chief programmer), who plans, coordinates, and reviews all technical activities of the team; *technical staff* (normally two to five people), who conduct analysis and development activities; and a *backup engineer,* who supports the senior engineer in his or her activities and can replace the senior engineer with minimum loss in project continuity. The chief programmer may be served by one or more specialists (e.g., telecommunications expert, database designer), support staff (e.g., technical writers, clerical personnel), and a software librarian.

As a counterpoint to the chief programmer team structure, Constantine's random paradigm [Con93] suggests a software team with creative independence whose approach to work might best be termed *innovative anarchy.* Although the free-spirited approach to software work has appeal, channeling creative energy into a high-performance team must be a central goal of a software engineering organization. To achieve a high-performance team:

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.

Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness. In their book, *Peopleware,* DeMarco and Lister [DeM98] discuss this issue:

> We tend to use the word team fairly loosely in the business world, calling any group of people assigned to work together a "team." But many of these groups just don't seem like

? What is a "jelled" team?

> teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell.*
>
> A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts. . . .
>
> Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success. . . . They don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got momentum.

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of eliteness" that makes them unique.

? Why is it that teams fail to jell?

But not all teams jell. In fact, many teams suffer from what Jackman [Jac98] calls "team toxicity." She defines five factors that "foster a potentially toxic team environment": (1) a frenzied work atmosphere, (2) high frustration that causes friction among team members, (3) a "fragmented or poorly coordinated" software process, (4) an unclear definition of roles on the software team, and (5) "continuous and repeated exposure to failure."

uote:

"Do or do not. There is no try."

Yoda from *Star Wars*

To avoid a frenzied work environment, the project manager should be certain that the team has access to all information required to do the job and that major goals and objectives, once defined, should not be modified unless absolutely necessary. A software team can avoid frustration if it is given as much responsibility for decision making as possible. An inappropriate process (e.g., unnecessary or burdensome work tasks or poorly chosen work products) can be avoided by understanding the product to be built, the people doing the work, and by allowing the team to select the process model. The team itself should establish its own mechanisms for accountability (technical reviews[3] are an excellent way to accomplish this) and define a series of corrective approaches when a member of the team fails to perform. And finally, the key to avoiding an atmosphere of failure is to establish team-based techniques for feedback and problem solving.

In addition to the five toxins described by Jackman, a software team often struggles with the differing human traits of its members. Some team members are extroverts; others are introverts. Some people gather information intuitively, distilling broad concepts from disparate facts. Others process information linearly, collecting and organizing minute details from the data provided. Some team members are comfortable making decisions only when a logical, orderly argument is presented. Others are intuitive, willing to make a decision based on "feel." Some practitioners want a detailed schedule populated by organized tasks that enable them to achieve closure for some element of a project. Others prefer a more spontaneous environment in which open issues are okay. Some work hard to get things done long before a milestone date, thereby avoiding stress as the date approaches, while others are

[3] Technical reviews are discussed in detail in Chapter 15.

energized by the rush to make a last-minute deadline. A detailed discussion of the psychology of these traits and the ways in which a skilled team leader can help people with opposing traits to work together is beyond the scope of this book.[4] However, it is important to note that recognition of human differences is the first step toward creating teams that jell.

24.2.4 Agile Teams

Over the past decade, agile software development (Chapter 3) has been suggested as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams, informal methods, minimal software engineering work products, and overall development simplicity.

The small, highly motivated project team, also called an *agile team,* adopts many of the characteristics of successful software project teams discussed in the preceding section and avoids many of the toxins that create problems. However, the agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team. Cockburn and Highsmith [Coc01a] note this when they write:

> If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy—"people trump process" is one way to say this. However, lack of user and executive support can kill a project—"politics trump people." Inadequate support can keep even good people from accomplishing the job.

KEY POINT

An agile team is a self-organizing team that has autonomy to plan and make technical decisions.

To make effective use of the competencies of each team member and to foster effective collaboration through a software project, agile teams are *self-organizing.* A self-organizing team does not necessarily maintain a single team structure but instead uses elements of Constantine's random, open, and synchronous paradigms discussed in Section 24.2.3.

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day.

Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self-organization and collaboration move the team toward a completed software increment.

[4] An excellent introduction to these issues as they relate to software project teams can be found in [Fer98].

24.2.5 Coordination and Communication Issues

Quote:

"Collective ownership is nothing more than an instantiation of the idea that products should be attributable to the [agile] team, not individuals who make up the team."

Jim Highsmith

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common, resulting in a continuing stream of changes that ratchets the project team. Interoperability has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software—scale, uncertainty, and interoperability—are facts of life. To deal with them effectively, you must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively non-interactive and impersonal communication channels" [Kra95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

SafeHome

Team Structure

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: Have you guys had a chance to look over the preliminary info on *SafeHome* that marketing has prepared?

Vinod (nodding and looking at his teammates): Yes. But we have a bunch of questions.

Doug: Let's hold on that for a moment. I'd like to talk about how we're going to structure the team, who's responsible for what . . .

Jamie: I'm really into the agile philosophy, Doug. I think we should be a self-organizing team.

Vinod: I agree. Given the tight time line and some of the uncertainty, and that fact that we're all really competent [laughs], that seems like the right way to go.

Doug: That's okay with me, but you guys know the drill.

Jamie (smiling and talking as if she was reciting something): "We make tactical decisions, about who does what and when, but it's our responsibility to get product out the door on time.

Vinod: And with quality.

Doug: Exactly. But remember there are constraints. Marketing defines the software increments to be produced—in consultation with us, of course.

Jamie: And?

Doug: And, we're going to use UML as our modeling approach.

Vinod: But keep extraneous documentation to an absolute minimum.

Doug: Who is the liaison with me?

Jamie: We decided that Vinod will be the tech lead—he's got the most experience, so Vinod is your liaison, but feel free to talk to any of us.

Doug (laughing): Don't worry, I will.

24.3 The Product

A software project manager is confronted with a dilemma at the very beginning of a software project. Quantitative estimates and an organized plan are required, but solid information is unavailable. A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or even months to complete. Worse, requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now!"

Like it or not, you must examine the product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

24.3.1 Software Scope

The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:

ADVICE

If you can't bound a characteristic of the software you intend to build, list the characteristic as a project risk (Chapter 25).

Context. How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?

Information objectives. What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, target environment, maximum allowable response time) are stated explicitly, constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in Java) are described.

24.3.2 Problem Decomposition

Problem decomposition, sometimes called *partitioning* or *problem elaboration,* is an activity that sits at the core of software requirements analysis (Chapters 6 and 7). During the scoping activity no attempt is made to fully decompose the problem. Rather, decomposition is applied in two major areas: (1) the functionality and content (information) that must be delivered and (2) the process that will be used to deliver it.

ADVICE

In order to develop a reasonable project plan, you must decompose the problem. This can be accomplished using a list of functions or with use cases.

Human beings tend to apply a divide-and-conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation (Chapter 26). Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Similarly, major content or data objects are decomposed into their constituent parts, providing a reasonable understanding of the information to be produced by the software.

As an example, consider a project that will build a new word-processing product. Among the unique features of the product are continuous voice as well as virtual keyboard input via a multitouch screen, extremely sophisticated "automatic copy edit" features, page layout capability, automatic indexing and table of contents, and others. The project manager must first establish a statement of scope that bounds these features (as well as other more mundane functions such as editing, file management, and document production). For example, will continuous voice input require that the product be "trained" by the user? Specifically, what capabilities will the copy edit feature provide? Just how sophisticated will the page layout capability be and will it encompass the capabilities implied by a multitouch screen?

As the statement of scope evolves, a first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing: (1) spell checking, (2) sentence grammar checking, (3) reference checking for large documents (e.g., Is a reference to a bibliography entry found in the list of entries in the bibliography?), (4) the implementation of a style sheet feature that imposed consistency across a document, and (5) section and chapter reference validation for large documents. Each of these features represents a subfunction to be implemented in software. Each can be further refined if the decomposition will make planning easier.

24.4 The Process

The framework activities (Chapter 2) that characterize the software process are applicable to all software projects. The problem is to select the process model that is appropriate for the software to be engineered by your project team.

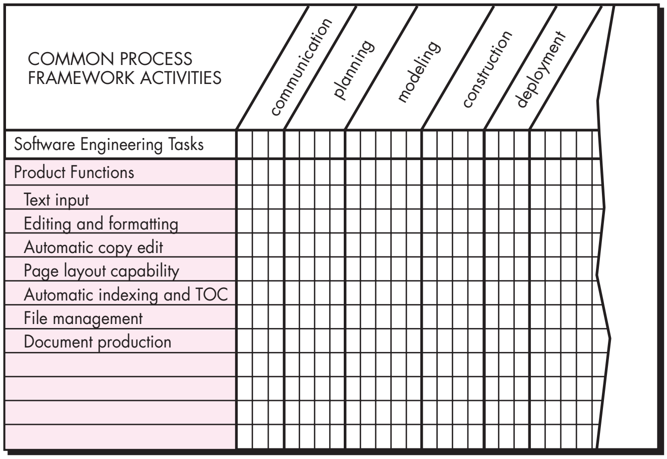
Your team must decide which process model is most appropriate for (1) the customers who have requested the product and the people who will do the work, (2) the characteristics of the product itself, and (3) the project environment in which the software team works. When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan, reflecting the work tasks required to populate the framework activities must be created. We explore these activities briefly in the sections that follow and present a more detailed view in Chapter 26.

24.4.1 Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by your team must pass through the set of framework activities that have been defined for your software organization.

FIGURE 24.1

Melding the problem and the process



COMMON PROCESS FRAMEWORK ACTIVITIES	communication	planning	modeling	construction	deployment
Software Engineering Tasks					
Product Functions					
Text input					
Editing and formatting					
Automatic copy edit					
Page layout capability					
Automatic indexing and TOC					
File management					
Document production					

Assume that the organization has adopted the generic framework activities—**communication, planning, modeling, construction,** and **deployment**—discussed in Chapter 2. The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 24.1 is created. Each major product function (the figure notes functions for the word-processing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.[5] The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each task. These activities are considered in Chapter 26.

24.4.2 Process Decomposition

KEY POINT

The process framework establishes a skeleton for project planning. It is adapted by allocating a task set that is appropriate to the project.

A software team should have a significant degree of flexibility in choosing the software process model that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics (e.g., uncertain requirements, breakthrough technology, difficult customers, significant reuse potential) will lead to the selection of other process models.[6]

[5] It should be noted that work tasks must be adapted to the specific needs of the project based on a number of adaptation criteria.

[6] Recall that project characteristics also have a strong bearing on the structure of the software team (Section 24.2.3).

Once the process model has been chosen, the process framework is adapted to it. In every case, the generic process framework discussed earlier can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models. The process framework is invariant and serves as the basis for all work performed by a software organization.

But actual work tasks do vary. Process decomposition commences when the project manager asks, "How do we accomplish this framework activity?" For example, a small, relatively simple project might require the following work tasks for the communication activity:

1. Develop list of clarification issues.
2. Meet with stakeholders to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project.

Now, consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the **communication**:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with all stakeholders.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, functional, and behavioral features of the software. Alternatively, develop use cases that describe the software from the user's point of view.
7. Review each mini-spec or use case for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document or collection of use cases with all concerned.
10. Modify the scoping document or use cases as required.

Both projects perform the framework activity that we call **communication**, but the first project team performs half as many software engineering work tasks as the second.

24.5 The Project

In order to manage a successful software project, you have to understand what can go wrong so that problems can be avoided. In an excellent paper on software projects, John Reel [Ree99] defines 10 signs that indicate that an information systems project is in jeopardy:

? **What are the signs that a software project is in jeopardy?**

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change [or are ill defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

Quote:

"We don't have time to stop for gas, we're already late."

M. Cleron

Jaded industry professionals often refer to the 90–90 rule when discussing particularly difficult software projects: The first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes another 90 percent of the allotted effort and time [Zah94]. The seeds that lead to the 90–90 rule are contained in the signs noted in the preceding list.

But enough negativity! How does a manager act to avoid the problems just noted? Reel [Ree99] suggests a five-part commonsense approach to software projects:

1. *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team (Section 24.2.3) and giving the team the autonomy, authority, and technology needed to do the job.
2. *Maintain momentum.* Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.[7]

[7] The implication of this statement is that bureaucracy is reduced to a minimum, extraneous meetings are eliminated, and dogmatic adherence to process and project rules is deemphasized. The team should be self-organizing and autonomous.

3. *Track progress.* For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity. In addition, software process and project measures (Chapter 25) can be collected and used to assess progress against averages developed for the software development organization.
4. *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components or patterns, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks (you'll need every minute).
5. *Conduct a postmortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

Quote:

"A project is like a road trip. Some projects are simple and routine, like driving to the store in broad daylight. But most projects worth doing are more like driving a truck off-road, in the mountains, at night."

Cem Kaner, James Bach, and Bret Pettichord

24.6 The W^{5}HH Principle

In an excellent paper on software process and projects, Barry Boehm [Boe96] states: "you need an organizing principle that scales down to provide simple [project] plans for simple projects." Boehm suggests an approach that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources. He calls it the *W^{5}HH Principle,* after a series of questions that lead to a definition of key project characteristics and the resultant project plan:

? **How do we define key project characteristics?**

Why is the system being developed? All stakeholders should assess the validity of business reasons for the software work. Does the business purpose justify the expenditure of people, time, and money?

What will be done? The task set required for the project is defined.

When will it be done? The team establishes a project schedule by identifying when project tasks are to be conducted and when milestones are to be reached.

Who is responsible for a function? The role and responsibility of each member of the software team is defined.

Where are they located organizationally? Not all roles and responsibilities reside within software practitioners. The customer, users, and other stakeholders also have responsibilities.

How will the job be done technically and managerially? Once product scope is established, a management and technical strategy for the project must be defined.

How much of each resource is needed? The answer to this question is derived by developing estimates (Chapter 26) based on answers to earlier questions.

Boehm's W^{5}HH Principle is applicable regardless of the size or complexity of a software project. The questions noted provide you and your team with an excellent planning outline.

24.7 Critical Practices

The Airlie Council[8] has developed a list of "critical software practices for performance-based management." These practices are "consistently used by, and considered critical by, highly successful software projects and organizations whose 'bottom line' performance is consistently much better than industry averages" [Air99].

Critical practices[9] include: metric-based project management (Chapter 25), empirical cost and schedule estimation (Chapters 26 and 27), earned value tracking (Chapter 27), defect tracking against quality targets (Chapters 14 though 16), and people aware management (Section 24.2). Each of these critical practices is addressed throughout Parts 3 and 4 of this book.

Software Tools

Software Tools for Project Managers

The "tools" listed here are generic and apply to a broad range of activities performed by project managers. Specific project management tools (e.g., scheduling tools, estimating tools, risk analysis tools) are considered in later chapters.

Representative Tools:[10]

The Software Program Manager's Network (**www.spmn.com**) has developed a simple tool called *Project Control Panel,* which provides project managers with an direct indication of project status. The tool has "gauges" much like a dashboard and is implemented with Microsoft Excel. It is available for download at **www.spmn.com/products_software.html**.

Ganthead.com (**www.gantthead.com/**) has developed a set of useful *checklists for project managers.*

Ittoolkit.com (**www.ittoolkit.com**) provides "a collection of planning guides, process templates and smart worksheets" available on CD-ROM.

[8] The Airlie Council was comprised of a team of software engineering experts chartered by the U.S. Department of Defense to help develop guidelines for best practices in software project management and software engineering. For more on best practices, see **www.swqual.com/newsletter/vol1/no3/vol1no3.html**.

[9] Only those critical practices associated with "project integrity" are noted here.

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

24.8 Summary

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the modeling, construction, and deployment of computer software.

Four P's have a substantial influence on software project management—people, product, process, and project. People must be organized into effective teams, motivated to do high-quality software work, and coordinated to achieve effective communication. Product requirements must be communicated from customer to developer, partitioned (decomposed) into their constituent parts, and positioned for work by the software team. The process must be adapted to the people and the product. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done. Finally, the project must be organized in a manner that enables the software team to succeed.

The pivotal element in all software projects is people. Software engineers can be organized in a number of different team structures that range from traditional control hierarchies to "open paradigm" teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general, technical reviews and informal person-to-person communication have the most value for practitioners.

The project management activity encompasses measurement and metrics, estimation and scheduling, risk analysis, tracking, and control. Each of these topics is considered in the chapters that follow.

Problems and Points to Ponder

24.1. Based on information contained in this chapter and your own experience, develop "ten commandments" for empowering software engineers. That is, make a list of 10 guidelines that will lead to software people who work to their full potential.

24.2. The Software Engineering Institute's People Capability Maturity Model (People-CMM) takes an organized look at "key practice areas" that cultivate good software people. Your instructor will assign you one KPA for analysis and summary.

24.3. Describe three real-life situations in which the customer and the end user are the same. Describe three situations in which they are different.

24.4. The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

24.5. Review a copy of Weinberg's book [Wei86], and write a two- or three-page summary of the issues that should be considered in applying the MOI model.

24.6. You have been appointed a project manager within an information systems organization. Your job is to build an application that is quite similar to others your team has built, although this one is larger and more complex. Requirements have been thoroughly documented by the customer. What team structure would you choose and why? What software process model(s) would you choose and why?

24.7. You have been appointed a project manager for a small software products company. Your job is to build a breakthrough product that combines virtual reality hardware with state-of-the-art software. Because competition for the home entertainment market is intense, there is significant pressure to get the job done. What team structure would you choose and why? What software process model(s) would you choose and why?

24.8. You have been appointed a project manager for a major software products company. Your job is to manage the development of the next-generation version of its widely used word-processing software. Because competition is intense, tight deadlines have been established and announced. What team structure would you choose and why? What software process model(s) would you choose and why?

24.9. You have been appointed a software project manager for a company that services the genetic engineering world. Your job is to manage the development of a new software product that will accelerate the pace of gene typing. The work is R&D oriented, but the goal is to produce a product within the next year. What team structure would you choose and why? What software process model(s) would you choose and why?

24.10. You have been asked to develop a small application that analyzes each course offered by a university and reports the average grade obtained in the course (for a given term). Write a statement of scope that bounds this problem.

24.11. Do a first-level functional decomposition of the page layout function discussed briefly in Section 24.3.2.

Further Readings and Information Sources

The Project Management Institute (*Guide to the Project Management Body of Knowledge,* PMI, 2001) covers all important aspects of project management. Bechtold (*Essentials of Software Project Management,* 2d ed., Management Concepts, 2007), Wysocki (*Effective Software Project Management,* Wiley, 2006), Stellman and Greene (*Applied Software Project Management,* O'Reilly, 2005), and Berkun (*The Art of Project Management,* O'Reilly, 2005) teach basic skills and provide detailed guidance for all software project management tasks. McConnell (*Professional Software Development,* Addison-Wesley, 2004) offers pragmatic advice for achieving "shorter schedules, higher quality products, and more successful projects." Henry (*Software Project Management,* Addison-Wesley, 2003) offers real-world advice that is useful for all project managers.

Tom DeMarco and his colleagues (*Adrenaline Junkies and Template Zombies,* Dorset House, 2008) have written an insightful treatment of the human patterns that are encountered in every software project. An excellent four-volume series written by Weinberg (*Quality Software Management,* Dorset House, 1992, 1993, 1994, 1996) introduces basic systems thinking and management concepts, explains how to use measurements effectively, and addresses "congruent action," the ability to establish "fit" between the manager's needs, the needs of technical staff, and the needs of the business. It will provide both new and experienced managers with useful information. Futrell and his colleagues (*Quality Software Project Management,* Prentice-Hall, 2002) present a voluminous treatment of project management. Brown and his colleagues (*Antipatterns in Project Management,* Wiley, 2000) discuss what not to do during the management of a software project.

Brooks (*The Mythical Man-Month,* Anniversary Edition, Addison-Wesley, 1995) has updated his classic book to provide new insight into software project and management issues. McConnell (*Software Project Survival Guide,* Microsoft Press, 1997) presents excellent pragmatic guidance for those who must manage software projects. Purba and Shah (*How to Manage a Successful Software Project,* 2d ed., Wiley, 2000) present a number of case studies that indicate why some projects succeed and others fail. Bennatan (*On Time Within Budget,* 3d ed., Wiley, 2000) presents useful tips and guidelines for software project managers. Weigers (*Practical Project Initiation,* Microsoft Press, 2007) provides practical guidelines for getting a software project off the ground successfully.

It can be argued that the most important aspect of software project management is people management. Cockburn (*Agile Software Development,* Addison-Wesley, 2002) presents one of the best discussions of software people written to date. DeMarco and Lister [DeM98] have written the definitive book on software people and software projects. In addition, the following books on this subject have been published in recent years and are worth examining:

Cantor, M., *Software Leadership: A Guide to Successful Software Development,* Addison-Wesley, 2001.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones,* Prentice Hall, 1999.

Constantine, L., *Peopleware Papers: Notes on the Human Side of Software,* Prentice Hall, 2001.

Garton, C., and K. Wegryn, *Managing Without Walls,* McPress, 2006.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, 1997.

Humphrey, W. S., *TSP-Coaching Development Teams,* Addison-Wesley, 2006.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development,* McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments,* IEEE Computer Society, 1998.

Peters, L., *Getting Results from Software Development Teams,* Microsoft Press, 2008.

Whitehead, R., *Leading a Software Development Team,* Addison-Wesley, 2001.

Even though they do not relate specifically to the software world and sometimes suffer from oversimplification and broad generalization, best-selling "management" books by Kanter (*Confidence,* Three Rivers Press, 2006), Covy (*The 8th Habit,* Free Press, 2004), Bossidy (*Execution: The Discipline of Getting Things Done,* Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century,* Harper Business, 1999), Buckingham and Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently,* Simon and Schuster, 1999), and Christensen (*The Innovator's Dilemma,* Harvard Business School Press, 1997) emphasize "new rules" defined by a rapidly changing economy. Older titles such as *Who Moved My Cheese?*, *The One-Minute Manager,* and *In Search of Excellence* continue to provide valuable insights that can help you to manage people and projects more effectively.

A wide variety of information sources on the software project management are available on the Internet. An up-to-date list of World Wide Web references relevant to software project management can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CHAPTER

Project Scheduling 27

Key Concepts

In the late 1960s, a bright-eyed young engineer was chosen to "write" a computer program for an automated manufacturing application. The reason for his selection was simple. He was the only person in his technical group who had attended a computer programming seminar. He knew the ins and outs of assembly language and FORTRAN but nothing about software engineering and even less about project scheduling and tracking.

His boss gave him the appropriate manuals and a verbal description of what had to be done. He was informed that the project must be completed in two months.

He read the manuals, considered his approach, and began writing code. After two weeks, the boss called him into his office and asked how things were going.

"Really great," said the young engineer with youthful enthusiasm. "This was much simpler than I thought. I'm probably close to 75 percent finished."

Quick Look

What is it? You've selected an appropriate process model, you've identified the software engineering tasks that have to be performed, you estimated the amount of work and the number of people, you know the deadline, you've even considered the risks. Now it's time to connect the dots. That is, you have to create a network of software engineering tasks that will enable you to get the job done on time. Once the network is created, you have to assign responsibility for each task, make sure it gets done, and adapt the network as risks become reality. In a nutshell, that's software project scheduling and tracking.

Who does it? At the project level, software project managers using information solicited from software engineers. At an individual level, software engineers themselves.

Why is it important? In order to build a complex system, many software engineering tasks occur in parallel, and the result of work performed during one task may have a profound effect on work to be conducted in another task. These interdependencies are very difficult to understand without a schedule. It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

What are the steps? The software engineering tasks dictated by the software process model are refined for the functionality to be built. Effort and duration are allocated to each task and a task network (also called an "activity network") is created in a manner that enables the software team to meet the delivery deadline established.

What is the work product? The project schedule and related information are produced.

How do I ensure that I've done it right? Proper scheduling requires that: (1) all tasks appear in the network, (2) effort and timing are intelligently allocated to each task, (3) interdependencies between tasks are properly indicated, (4) resources are allocated for the work to be done, and (5) closely spaced milestones are provided so that progress can be tracked.

time-line charts 732
tracking 734
work breakdown 732

The boss smiled and encouraged the young engineer to keep up the good work. They planned to meet again in a week's time.

A week later the boss called the engineer into his office and asked, "Where are we?"

"Everything's going well," said the youngster, "but I've run into a few small snags. I'll get them ironed out and be back on track soon."

"How does the deadline look?" the boss asked.

"No problem," said the engineer. "I'm close to 90 percent complete."

If you've been working in the software world for more than a few years, you can finish the story. It'll come as no surprise that the young engineer[1] stayed 90 percent complete for the entire project duration and finished (with the help of others) only one month late.

This story has been repeated tens of thousands of times by software developers during the past five decades. The big question is why?

27.1 Basic Concepts

Although there are many reasons why software is delivered late, most can be traced to one or more of the following root causes:

- An unrealistic deadline established by someone outside the software team and forced on managers and practitioners.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Quote:

"Excessive or irrational schedules are probably the single most destructive influence in all of software."

Capers Jones

Aggressive (read "unrealistic") deadlines are a fact of life in the software business. Sometimes such deadlines are demanded for reasons that are legitimate, from the point of view of the person who sets the deadline. But common sense says that legitimacy must also be perceived by the people doing the work.

Napoleon once said: "Any commander-in-chief who undertakes to carry out a plan which he considers defective is at fault; he must put forth his reasons, insist on the plan being changed, and finally tender his resignation rather than be the instrument of his army's downfall." These are strong words that many software project managers should ponder.

[1] In case you were wondering, this story is autobiographical.

The estimation activities discussed in Chapter 26 and the scheduling techniques described in this chapter are often implemented under the constraint of a defined deadline. If best estimates indicate that the deadline is unrealistic, a competent project manager should "protect his or her team from undue [schedule] pressure . . . [and] reflect the pressure back to its originators" [Pag85].

Quote:

"I love deadlines. I like the whooshing sound they make as they fly by."

Douglas Adams

To illustrate, assume that your software team has been asked to build a real-time controller for a medical diagnostic instrument that is to be introduced to the market in nine months. After careful estimation and risk analysis (Chapter 28), you come to the conclusion that the software, as requested, will require 14 calendar months to create with available staff. How should you proceed?

It is unrealistic to march into the customer's office (in this case the likely customer is marketing/sales) and demand that the delivery date be changed. External market pressures have dictated the date, and the product must be released. It is equally foolhardy to refuse to undertake the work (from a career standpoint). So, what to do? I recommend the following steps in this situation:

1. Perform a detailed estimate using historical data from past projects. Determine the estimated effort and duration for the project.
2. Using an incremental process model (Chapter 2), develop a software engineering strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.

? What should you do when management demands a deadline that is impossible?

3. Meet with the customer and (using the detailed estimate), explain why the imposed deadline is unrealistic. Be certain to note that all estimates are based on performance on past projects. Also be certain to indicate the percent improvement that would be required to achieve the deadline as it currently exists.[2] The following comment is appropriate:

   > I think we may have a problem with the delivery date for the XYZ controller software. I've given each of you an abbreviated breakdown of development rates for past software projects and an estimate that we've done a number of different ways. You'll note that I've assumed a 20 percent improvement in past development rates, but we still get a delivery date that's 14 calendar months rather than 9 months away.

4. Offer the incremental development strategy as an alternative:

   > We have a few options, and I'd like you to make a decision based on them. First, we can increase the budget and bring on additional resources so that we'll have a shot at getting this job done in nine months. But understand that this will increase the risk of poor quality due to the tight time line.[3] Second, we can remove a number of the software functions and capabilities that you're requesting. This will make the preliminary

[2] If the required improvement is 10 to 25 percent, it may actually be possible to get the job done. But, more likely, the required improvement in team performance will be greater than 50 percent. This is an unrealistic expectation.

[3] You might also add that increasing the number of people does not reduce calendar time proportionally.

> version of the product somewhat less functional, but we can announce all functionality and then deliver over the 14-month period. Third, we can dispense with reality and wish the project complete in nine months. We'll wind up with nothing that can be delivered to a customer. The third option, I hope you'll agree, is unacceptable. Past history and our best estimates say that it is unrealistic and a recipe for disaster.

There will be some grumbling, but if a solid estimate based on good historical data is presented, it's likely that negotiated versions of option 1 or 2 will be chosen. The unrealistic deadline evaporates.

27.2 Project Scheduling

Fred Brooks was once asked how software projects fall behind schedule. His response was as simple as it was profound: "One day at a time."

The reality of a technical project (whether it involves building a hydroelectric plant or developing an operating system) is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other tasks lie on the "critical path." If these "critical" tasks fall behind schedule, the completion date of the entire project is put into jeopardy.

ADVICE

The tasks required to achieve a project manager's objective should not be performed manually. There are many excellent scheduling tools. Use them.

As a project manager, your objective is to define all project tasks, build a network that depicts their interdependencies, identify the tasks that are critical within the network, and then track their progress to ensure that delay is recognized "one day at a time." To accomplish this, you must have a schedule that has been defined at a degree of resolution that allows progress to be monitored and the project to be controlled.

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software actions and tasks (required to accomplish an activity) are identified and scheduled.

Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second.

Quote:

"Overly optimistic scheduling doesn't result in shorter actual schedules, it results in longer ones."

Steve McConnell

27.2.1 Basic Principles

Like all other areas of software engineering, a number of basic principles guide software project scheduling:

KEY POINT

When you develop a schedule, compartmentalize the work, note task interdependencies, allocate effort and time to each task, and define responsibilities, outcomes, and milestones.

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort[4]). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 15) and has been approved.

Each of these principles is applied as the project schedule evolves.

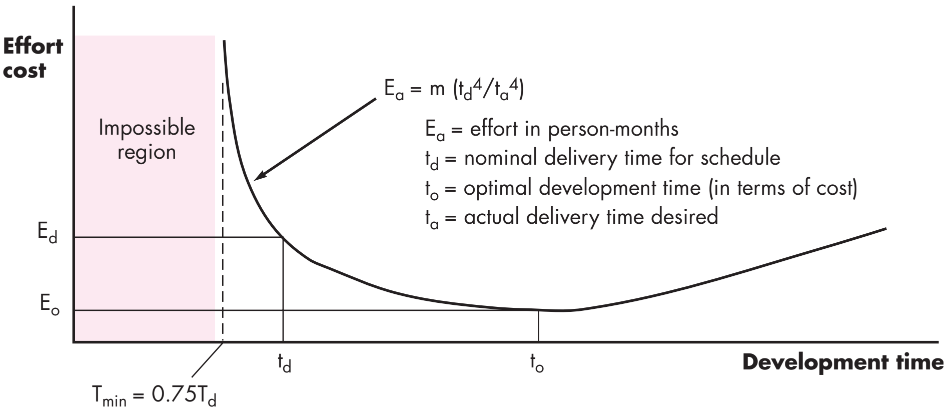
27.2.2 The Relationship Between People and Effort

In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved. (We can rarely afford the luxury of approaching a 10 person-year effort with one person working for 10 years!)

[4] In reality, less than three person-days of effort are available because of unrelated meetings, sickness, vacation, and a variety of other reasons. For our purposes, however, we assume 100 percent availability.

FIGURE 27.1

The relationship between effort and delivery time



ADVICE

If you must add people to a late project, be sure that you've assigned them work that is highly compartmentalized.

There is a common myth that is still believed by many managers who are responsible for software development projects: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind.

In addition to the time it takes to learn the system, more people increase the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

Over the years, empirical data and theoretical analysis have demonstrated that project schedules are elastic. That is, it is possible to compress a desired project completion date (by adding additional resources) to some extent. It is also possible to extend a completion date (by reducing the number of resources).

KEY POINT

If delivery can be delayed, the PNR curve indicates that project costs can be reduced substantially.

The *Putnam-Norden-Rayleigh (PNR) Curve*[5] provides an indication of the relationship between effort applied and delivery time for a software project. A version of the curve, representing project effort as a function of delivery time, is shown in Figure 27.1. The curve indicates a minimum value t_o that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). As we move left of t_o (i.e., as we try to accelerate delivery), the curve rises nonlinearly.

As an example, we assume that a project team has estimated a level of effort E_d will be required to achieve a nominal delivery time t_d that is optimal in terms of

[5] Original research can be found in [Nor70] and [Put78].

schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of t_d. In fact, the PNR curve indicates that the project delivery time cannot be compressed much beyond $0.75t_d$. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes very high. The PNR curve also indicates that the lowest cost delivery option, $t_o = 2t_d$. The implication here is that delaying project delivery can reduce costs significantly. Of course, this must be weighed against the business cost associated with the delay.

The software equation [Put92] introduced in Chapter 26 is derived from the PNR curve and demonstrates the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code (source statements), *L,* is related to effort and development time by the equation:

$$L = P \times E^{1/3}t^{4/3}$$

where *E* is development effort in person-months, *P* is a productivity parameter that reflects a variety of factors that lead to high-quality software engineering work (typical values for *P* range between 2000 and 12,000), and *t* is the project duration in calendar months.

ADVICE

As the project deadline becomes tighter and tighter, you reach a point at which the work cannot be completed on schedule, regardless of the number of people doing the work. Face reality and define a new delivery date.

Rearranging this software equation, we can arrive at an expression for development effort *E*:

$$E = \frac{L^3}{P^3t^4} \tag{27.1}$$

where *E* is the effort expended (in person-years) over the entire life cycle for software development and maintenance and *t* is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year).

This leads to some interesting results. Consider a complex, real-time software project estimated at 33,000 LOC, 12 person-years of effort. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years, the highly nonlinear nature of the model described in Equation (27.1) yields:

$$E = \frac{L^3}{P^3t^4} \sim 3.8 \text{ person-years}$$

This implies that, by extending the end date by six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear: benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

27.2.3 Effort Distribution

Each of the software project estimation techniques discussed in Chapter 26 leads to estimates of work units (e.g., person-months) required to complete software development. A recommended distribution of effort across the software process is often referred to as the *40–20–40 rule.* Forty percent of all effort is allocated to front-end analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized.

? How should effort be distributed across the software process workflow?

This effort distribution should be used as a guideline only.[6] The characteristics of each project dictate the distribution of effort. Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk. Customer communication and requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity. A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

27.3 Defining a Task Set for the Software Project

Regardless of the process model that is chosen, the work that a software team performs is achieved through a set of tasks that enable you to define, develop, and ultimately support computer software. No single task set is appropriate for all projects. The set of tasks that would be appropriate for a large, complex system would likely be perceived as overkill for a small, relatively simple software product. Therefore, an effective software process should define a collection of task sets, each designed to meet the needs of different types of projects.

As I noted in Chapter 2, a task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project. The task set must provide enough discipline to achieve high software quality. But, at the same time, it must not burden the project team with unnecessary work.

In order to develop a project schedule, a task set must be distributed on the project time line. The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work. Although it is difficult

[6] Today, the 40-20-40 rule is under attack. Some believe that more than 40 percent of overall effort should be expended during analysis and design. On the other hand, some proponents of agile development (Chapter 3) argue that less time should be expended "up front" and that a team should move quickly to construction.

to develop a comprehensive taxonomy of software project types, most software organizations encounter the following projects:

WebRef

An adaptable process model (APM) has been developed to assist in defining task sets for various software projects. A complete description of the APM can be found at **www.rspa.com/apm**.

1. *Concept development projects* that are initiated to explore some new business concept or application of some new technology.
2. *New application development* projects that are undertaken as a consequence of a specific customer request.
3. *Application enhancement* projects that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.
4. *Application maintenance projects* that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.
5. *Reengineering projects* that are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

Even within a single project type, many factors influence the task set to be chosen. These include [Pre05]: size of the project, number of potential users, mission criticality, application longevity, stability of requirements, ease of customer/developer communication, maturity of applicable technology, performance constraints, embedded and nonembedded characteristics, project staff, and reengineering factors. When taken in combination, these factors provide an indication of the *degree of rigor* with which the software process should be applied.

27.3.1 A Task Set Example

Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer (e.g., marketing) believes that potential benefit exists. Concept development projects are approached by applying the following actions:

1.1 **Concept scoping** determines the overall scope of the project.

1.2 **Preliminary concept planning** establishes the organization's ability to undertake the work implied by the project scope.

1.3 **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of the project scope.

1.4 **Proof of concept** demonstrates the viability of a new technology in the software context.

1.5 **Concept implementation** implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

1.6 **Customer reaction** to the concept solicits feedback on a new technology concept and targets specific customer applications.

A quick scan of these actions should yield few surprises. In fact, the software engineering flow for concept development projects (and for all other types of projects as well) is little more than common sense.

27.3.2 Refinement of Software Engineering Actions

The software engineering actions described in the preceding section may be used to define a macroscopic schedule for a project. However, the macroscopic schedule must be refined to create a detailed project schedule. Refinement begins by taking each action and decomposing it into a set of tasks (with related work products and milestones).

As an example of task decomposition, consider Action 1.1, Concept Scoping. Task refinement can be accomplished using an outline format, but in this book, a process design language approach is used to illustrate the flow of the concept scoping action:

```
Task definition: Action 1.1 Concept Scoping
1.1.1   Identify need, benefits and potential customers;
1.1.2   Define desired output/control and input events that drive the application;
          Begin Task 1.1.2
          1.1.2.1 TR: Review written description of need[7]
          1.1.2.2 Derive a list of customer visible outputs/inputs
          1.1.2.3 TR: Review outputs/inputs with customer and revise as required; endtask
          Task 1.1.2
1.1.3   Define the functionality/behavior for each major function;
          Begin Task 1.1.3
          1.1.3.1 TR: Review output and input data objects derived in task 1.1.2;
          1.1.3.2 Derive a model of functions/behaviors;
          1.1.3.3 TR: Review functions/behaviors with customer and revise as required;
          endtask Task 1.1.3
1.1.4   Isolate those elements of the technology to be implemented in software;
1.1.5   Research availability of existing software;
1.1.6   Define technical feasibility;
1.1.7   Make quick estimate of size;
1.1.8   Create a scope definition;
endtask definition: Action 1.1
```

The tasks and subtasks noted in the process design language refinement form the basis for a detailed schedule for the concept scoping action.

[7] TR indicates that a technical review (Chapter 15) is to be conducted.

27.4 Defining a Task Network

KEY POINT

The task network is a useful mechanism for depicting intertask dependencies and determining the critical path.

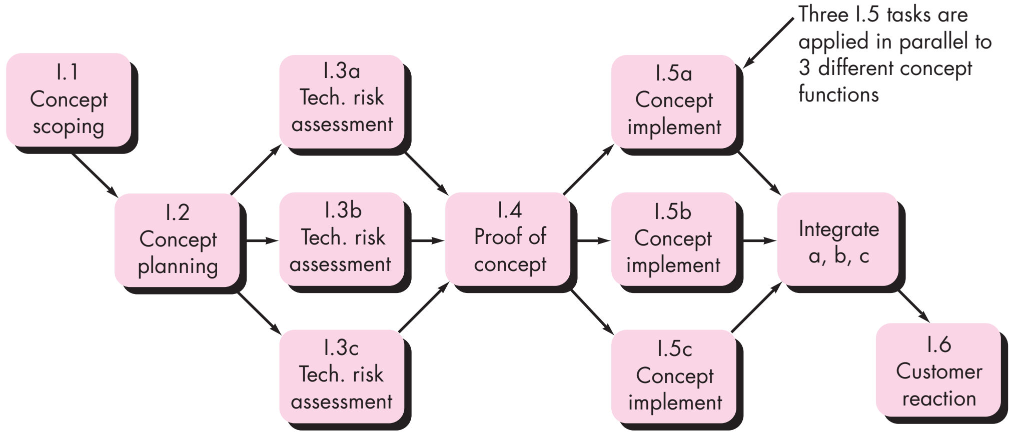
Individual tasks and subtasks have interdependencies based on their sequence. In addition, when more than one person is involved in a software engineering project, it is likely that development activities and tasks will be performed in parallel. When this occurs, concurrent tasks must be coordinated so that they will be complete when later tasks require their work product(s).

A *task network,* also called an *activity network,* is a graphic representation of the task flow for a project. It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool. In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering actions. Figure 27.2 shows a schematic task network for a concept development project.

The concurrent nature of software engineering actions leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, you should determine intertask dependencies to ensure continuous progress toward completion. In addition, you should be aware of those tasks that lie on the *critical path.* That is, tasks that must be completed on schedule if the project as a whole is to be completed on schedule. These issues are discussed in more detail later in this chapter.

It is important to note that the task network shown in Figure 27.2 is macroscopic. In a detailed task network (a precursor to a detailed schedule), each action shown in the figure would be expanded. For example, Task 1.1 would be expanded to show all tasks detailed in the refinement of Actions 1.1 shown in Section 27.3.2.

FIGURE 27.2 A task network for concept development



27.5 Scheduling

> **Quote:**
>
> "*All* we have to decide is what to do with the time that is given to us."
>
> **Gandalf in *The Lord of the Rings: Fellowship of the Rings***

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.

Program evaluation and review technique (PERT) and the *critical path method* (CPM) are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities: estimates of effort, a decomposition of the product function, the selection of the appropriate process model and task set, and decomposition of the tasks that are selected.

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow you to (1) determine the critical path—the chain of tasks that determines the duration of the project, (2) establish "most likely" time estimates for individual tasks by applying statistical models, and (3) calculate "boundary times" that define a time "window" for a particular task.

27.5.1 Time-Line Charts

When creating a software project schedule, you begin with a set of tasks (the work breakdown structure). If automated tools are used, the work breakdown is input as

Software Tools

Project Scheduling

Objective: The objective of project scheduling tools is to enable a project manager to define work tasks; establish their dependencies; assign human resources to tasks; and develop a variety of graphs, charts, and tables that aid in tracking and control of the software project.

Mechanics: In general, project scheduling tools require the specification of a work breakdown structure of tasks or the generation of a task network. Once the task breakdown (an outline) or network is defined, start and end dates, human resources, hard deadlines, and other data are attached to each. The tool then generates a variety of time-line charts and other tables that enable a manager to assess the task flow of a project. These data can be updated continually as the project is under way.

Representative Tools:[8]

AMS Realtime, developed by Advanced Management Systems (**www.amsusa.com**), provides scheduling capabilities for projects of all sizes and types.

Microsoft Project, developed by Microsoft (**www.microsoft.com**), is the most widely used PC-based project scheduling tool.

4C, developed by *4C Systems* (**www.4csys.com**), supports all aspects of project planning including scheduling.

A comprehensive list of project management software vendors and products can be found at **www.infogoal.com/pmc/pmcswr.htm**.

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

a task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.

KEY POINT

A time-line chart enables you to determine what tasks will be conducted at a given point in time.

As a consequence of this input, a *time-line chart,* also called a *Gantt chart,* is generated. A time-line chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.

Figure 27.3 illustrates the format of a time-line chart. It depicts a part of a software project schedule that emphasizes the concept scoping task for a word-processing (WP) software product. All project tasks (for concept scoping) are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.

Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce *project tables*—a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information (Figure 27.4). Used in conjunction with the time-line chart, project tables enable you to track progress.

FIGURE 27.3 An example time-line chart

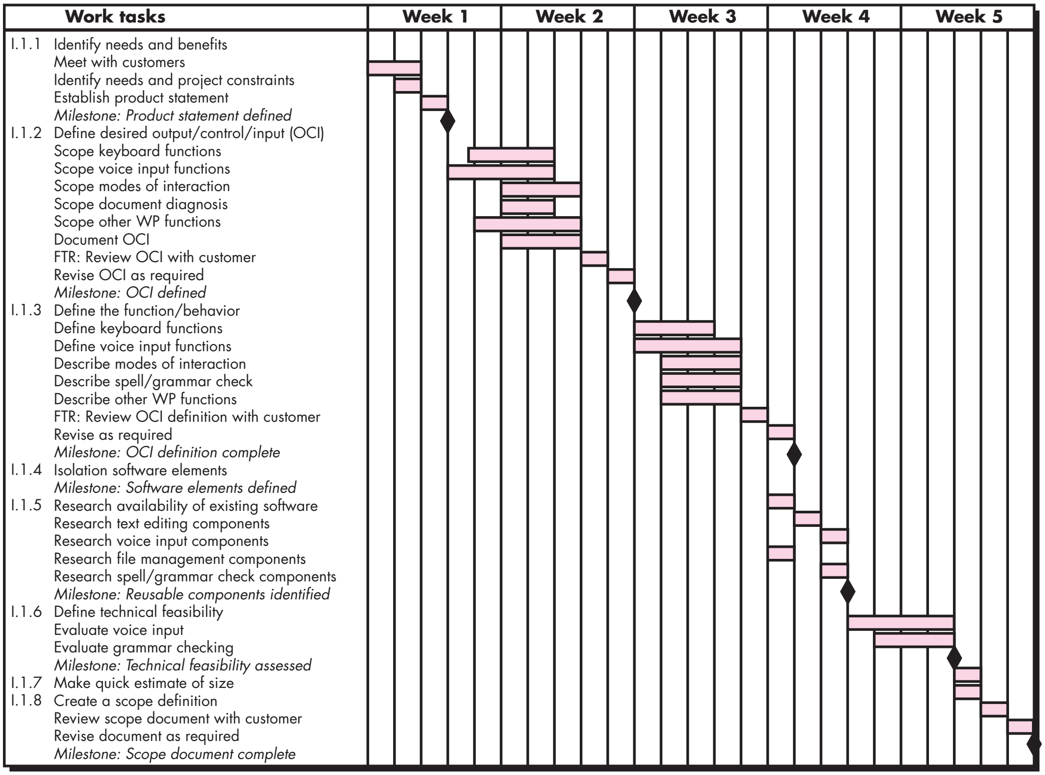
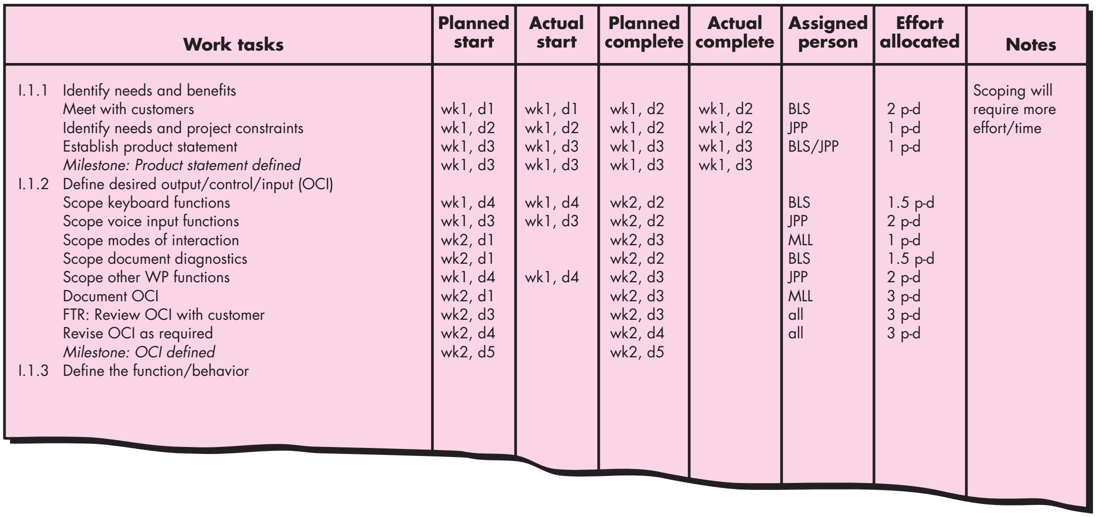


FIGURE 27.4 An example project table



Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
1.1.1 Identify needs and benefits							Scoping will
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	require more
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	effort/time
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
1.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
1.1.3 Define the function/behavior							

27.5.2 Tracking the Schedule

If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems
- Evaluating the results of all reviews conducted throughout the software engineering process
- Determining whether formal project milestones (the diamonds shown in Figure 27.3) have been accomplished by the scheduled date
- Comparing the actual start date to the planned start date for each project task listed in the resource table (Figure 27.4)
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon
- Using earned value analysis (Section 27.6) to assess progress quantitatively

uote:

"The basic rule of software status reporting can be summarized in a single phrase: 'No surprises'."

Capers Jones

In reality, all of these tracking techniques are used by experienced project managers.

Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff. If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems

ADVICE

The best indication of progress is the completion and successful review of a defined software work product.

occur, you must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed, additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.

When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called *time-boxing* [Jal04]. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore, an incremental software paradigm (Chapter 2) is chosen, and a schedule is derived for each incremental delivery.

KEY POINT

When the defined completion date of a time-boxed task is reached, work ceases for that task and the next task begins.

The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A "box" is put around each task. When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

The initial reaction to the time-boxing approach is often negative: "If the work isn't finished, how can we proceed?" The answer lies in the way work is accomplished. By the time the time-box boundary is encountered, it is likely that 90 percent of the task has been completed.[9] The remaining 10 percent, although important, can (1) be delayed until the next increment or (2) be completed later if required. Rather than becoming "stuck" on a task, the project proceeds toward the delivery date.

27.5.3 Tracking Progress for an OO Project

Although an iterative model is the best framework for an OO project, task parallelism makes project tracking difficult. You may have difficulty establishing meaningful milestones for an OO project because a number of different things are happening at once. In general, the following major milestones can be considered "completed" when the criteria noted have been met.

Technical milestone: OO analysis completed

- All classes and the class hierarchy have been defined and reviewed.
- Class attributes and operations associated with a class have been defined and reviewed.
- Class relationships (Chapter 6) have been established and reviewed.
- A behavioral model (Chapter 7) has been created and reviewed.
- Reusable classes have been noted.

Technical milestone: OO design completed

- The set of subsystems has been defined and reviewed.
- Classes are allocated to subsystems and reviewed.
- Task allocation has been established and reviewed.

[9] A cynic might recall the saying: "The first 90 percent of the system takes 90 percent of the time; the remaining 10 percent of the system takes 90 percent of the time."

- Responsibilities and collaborations have been identified.
- Attributes and operations have been designed and reviewed.
- The communication model has been created and reviewed.

Technical milestone: OO programming completed

- Each new class has been implemented in code from the design model.
- Extracted classes (from a reuse library) have been implemented.
- Prototype or increment has been built.

Technical milestone: OO testing

- The correctness and completeness of OO analysis and design models has been reviewed.
- A class-responsibility-collaboration network (Chapter 6) has been developed and reviewed.
- Test cases are designed, and class-level tests (Chapter 19) have been conducted for each class.
- Test cases are designed, and cluster testing (Chapter 19) is completed and the classes are integrated.
- System-level tests have been completed.

ADVICE

Debugging and testing occur in concert with one another. The status of debugging is often assessed by considering the type and number of "open" errors (bugs).

Recalling that the OO process model is iterative, each of these milestones may be revisited as different increments are delivered to the customer.

27.5.4 Scheduling for WebApp Projects

WebApp project scheduling distributes estimated effort across the planned time line (duration) for building each WebApp increment. This is accomplished by allocating the effort to specific tasks. It is important to note, however, that the overall WebApp schedule evolves over time. During the first iteration, a macroscopic schedule is developed. This type of schedule identifies all WebApp increments and projects the dates on which each will be deployed. As the development of an increment gets under way, the entry for the increment on the macroscopic schedule is refined into a detailed schedule. Here, specific development tasks (required to accomplish an activity) are identified and scheduled.

As an example of macroscopic scheduling, consider the **SafeHomeAssured.com** WebApp. Recalling earlier discussions of **SafeHomeAssured.com**, seven increments can be identified for the Web-based component of the project:

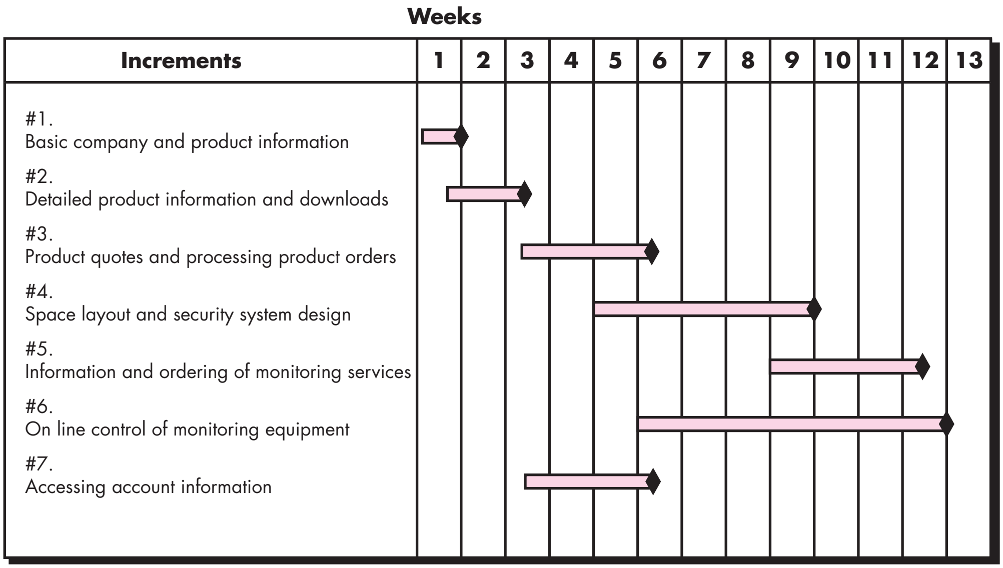
Increment 1: Basic company and product information

Increment 2: Detailed product information and downloads

Increment 3: Product quotes and processing product orders

Increment 4: Space layout and security system design

FIGURE 27.5 Time line for macroscopic project schedule



Increment 5: Information and ordering of monitoring services

Increment 6: Online control of monitoring equipment

Increment 7: Accessing account information

The team consults and negotiates with stakeholders and develops a *preliminary* deployment schedule for all seven increments. A time-line chart for this schedule is illustrated in Figure 27.5.

It is important to note that the deployment dates (represented by diamonds on the time-line chart) are preliminary and may change as more detailed scheduling of the increments occurs. However, this macroscopic schedule provides management with an indication of when content and functionality will be available and when the entire project will be completed. As a preliminary estimate, the team will work to deploy all increments with a 12-week time line. It's also worth noting that some of the increments will be developed in parallel (e.g., increments 3, 4, 6 and 7). This assumes that the team will have sufficient people to do this parallel work.

Once the macroscopic schedule has been developed, the team is ready to schedule work tasks for a specific increment. To accomplish this, you can use a generic process framework that is applicable for all WebApp increments. A *task list* is created by using the generic tasks derived as part of the framework as a starting point and then adapting these by considering the content and functions to be derived for a specific WebApp increment.

Each framework action (and its related tasks) can be adapted in one of four ways: (1) a task is applied as is, (2) a task is eliminated because it is not necessary for the increment, (3) a new (custom) task is added, and (4) a task is refined (elaborated) into a number of named subtasks that each becomes part of the schedule.

To illustrate, consider a generic *design modeling* action for WebApps that can be accomplished by applying some or all of the following tasks:

- Design the aesthetic for the WebApp.
- Design the interface.
- Design the navigation scheme.
- Design the WebApp architecture.
- Design the content and the structure that supports it.
- Design functional components.
- Design appropriate security and privacy mechanisms.
- Review the design.

As an example, consider the generic task *Design the Interface* as it is applied to the fourth increment of **SafeHomeAssured.com**. Recall that the fourth increment implements the content and function for describing the living or business space to be secured by the *SafeHome* security system. Referring to Figure 27.5, the fourth increment commences at the beginning of the fifth week and terminates at the end of the ninth week.

There is little question that the *Design the Interface* task must be conducted. The team recognizes that the interface design is pivotal to the success of the increment and decides to refine (elaborate) the task. The following subtasks are derived for the *Design the Interface* task for the fourth increment:

- Develop a sketch of the page layout for the space design page.
- Review layout with stakeholders.
- Design space layout navigation mechanisms.
- Design "drawing board" layout.[10]
- Develop procedural details for the graphical wall layout function.
- Develop procedural details for the wall length computation and display function.
- Develop procedural details for the graphical window layout function.
- Develop procedural details for the graphical door layout function.
- Design mechanisms for selecting security system components (sensors, cameras, microphones, etc.).

[10] At this stage, the team envisions creating the space by literally drawing the walls, windows, and doors using graphical functions. Wall lines will "snap" onto grip points. Dimensions of the wall will be displayed automatically. Windows and doors will be positioned graphically. The end user can also select specific sensors, cameras, etc., and position them once the space has been defined.

- Develop procedural details for the graphical layout of security system components.
- Conduct pair walkthroughs as required.

These tasks become part of the increment schedule for the fourth WebApp increment and are allocated over the increment development schedule. They can be input to scheduling software and used for tracking and control.

SafeHome

Tracking the Schedule

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug (glancing at a PowerPoint slide): The schedule for the first *SafeHome* increment seems reasonable, but we're going to have trouble tracking progress.

Vinod (a concerned look on his face): Why? We have tasks scheduled on a daily basis, plenty of work products, and we've been sure that we're not overallocating resources.

Doug: All good, but how do we know when the requirements model for the first increment is complete?

Jamie: Things are iterative, so that's difficult.

Doug: I understand that, but . . . well, for instance, take "analysis classes defined." You indicated that as a milestone.

Vinod: We have.

Doug: Who makes that determination?

Jamie (aggravated): They're done when they're done.

Doug: That's not good enough, Jamie. We have to schedule TRs [technical reviews, Chapter 15], and you haven't done that. The successful completion of a review on the analysis model, for instance, is a reasonable milestone. Understand?

Jamie (frowning): Okay, back to the drawing board.

Doug: It shouldn't take more than an hour to make the corrections . . . everyone else can get started now.

27.6 Earned Value Analysis

KEY POINT

Earned value provides a quantitative indication of progress.

In Section 27.5, I discussed a number of qualitative approaches to project tracking. Each provides the project manager with an indication of progress, but an assessment of the information provided is somewhat subjective. It is reasonable to ask whether there is a quantitative technique for assessing progress as the software team progresses through the work tasks allocated to the project schedule. In fact, a technique for performing quantitative analysis of progress does exist. It is called *earned value analysis* (EVA). Humphrey [Hum95] discusses earned value in the following manner:

> The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total.

Stated even more simply, earned value is a measure of progress. It enables you to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling. In fact, Fleming and Koppleman [Fle98] argue that earned value analysis "provides accurate and reliable readings of performance from as early as 15 percent into the project." To determine the earned value, the following steps are performed:

? How do I compute earned value and use it to assess progress?

1. The *budgeted cost of work scheduled* (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, BCWS_i is the effort planned for work task *i.* To determine progress at a given point along the project schedule, the value of BCWS is the sum of the BCWS_i values for all work tasks that should have been completed by that point in time on the project schedule.
2. The BCWS values for all work tasks are summed to derive the *budget at completion* (BAC). Hence,

   $\text{BAC} = \Sigma\ (\text{BCWS}_k)$ for all tasks k
3. Next, the value for *budgeted cost of work performed* (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Wilkens [Wil99] notes that "the distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed." Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

$$\text{Schedule performance index, SPI} = \frac{\text{BCWP}}{\text{BCWS}}$$

$$\text{Schedule variance, SV} = \text{BCWP} - \text{BCWS}$$

WebRef
A wide array of earned value analysis resources can be found at **www.acq.osd.mil/pm/**.

SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.

$$\text{Percent scheduled for completion} = \frac{\text{BCWS}}{\text{BAC}}$$

provides an indication of the percentage of work that should have been completed by time *t.*

$$\text{Percent complete} = \frac{\text{BCWP}}{\text{BAC}}$$

provides a quantitative indication of the percent of completeness of the project at a given point in time *t.*

It is also possible to compute the *actual cost of work performed* (ACWP). The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute

$$\text{Cost performance index, CPI} = \frac{\text{BCWP}}{\text{ACWP}}$$

$$\text{Cost variance, CV} = \text{BCWP} - \text{ACWP}$$

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project.

Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.

27.7 Summary

Scheduling is the culmination of a planning activity that is a primary component of software project management. When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

Scheduling begins with process decomposition. The characteristics of the project are used to adapt an appropriate task set for the work to be done. A task network depicts each engineering task, its dependency on other tasks, and its projected duration. The task network is used to compute the critical path, a time-line chart, and a variety of project information. Using the schedule as a guide, you can track and control each step in the software process.

Problems and Points to Ponder

27.1. "Unreasonable" deadlines are a fact of life in the software business. How should you proceed if you're faced with one?

27.2. What is the difference between a macroscopic schedule and a detailed schedule? Is it possible to manage a project if only a macroscopic schedule is developed? Why?

27.3. Is there ever a case where a software project milestone is not tied to a review? If so, provide one or more examples.

27.4. "Communication overhead" can occur when multiple people work on a software project. The time spent communicating with others reduces individual productively (LOC/month), and the result can be less productivity for the team. Illustrate (quantitatively) how engineers who are well versed in good software engineering practices and use technical reviews can increase the production rate of a team (when compared to the sum of individual production rates). Hint: You can assume that reviews reduce rework and that rework can account for 20 to 40 percent of a person's time.

27.5. Although adding people to a late software project can make it later, there are circumstances in which this is not true. Describe them.

27.6. The relationship between people and time is highly nonlinear. Using Putnam's software equation (described in Section 27.2.2), develop a table that relates number of people to project duration for a software project requiring 50,000 LOC and 15 person-years of effort (the productivity parameter is 5000 and $B = 0.37$). Assume that the software must be delivered in 24 months plus or minus 12 months.

27.7. Assume that you have been contracted by a university to develop an online course registration system (OLCRS). First, act as the customer (if you're a student, that should be easy!) and specify the characteristics of a good system. (Alternatively, your instructor will provide you with a set of preliminary requirements for the system.) Using the estimation methods discussed in Chapter 26, develop an effort and duration estimate for OLCRS. Suggest how you would:

a. Define parallel work activities during the OLCRS project.

b. Distribute effort throughout the project.

c. Establish milestones for the project.

27.8. Select an appropriate task set for the OLCRS project.

27.9. Define a task network for OLCRS described in Problem 27.7, or alternatively, for another software project that interests you. Be sure to show tasks and milestones and to attach effort and duration estimates to each task. If possible, use an automated scheduling tool to perform this work.

27.10. If an automated scheduling tool is available, determine the critical path for the network defined in Problem 27.9.

27.11. Using a scheduling tool (if available) or paper and pencil (if necessary), develop a timeline chart for the OLCRS project.

27.12. Assume you are a software project manager and that you've been asked to compute earned value statistics for a small software project. The project has 56 planned work tasks that are estimated to require 582 person-days to complete. At the time that you've been asked to do the earned value analysis, 12 tasks have been completed. However the project schedule indicates that 15 tasks should have been completed. The following scheduling data (in person-days) are available:

Task	Planned Effort	Actual Effort
1	12.0	12.5
2	15.0	11.0
3	13.0	17.0
4	8.0	9.5
5	9.5	9.0
6	18.0	19.0
7	10.0	10.0
8	4.0	4.5
9	12.0	10.0
10	6.0	6.5
11	5.0	4.0
12	14.0	14.5
13	16.0	—
14	6.0	—
15	8.0	—

Compute the SPI, schedule variance, percent scheduled for completion, percent complete, CPI, and cost variance for the project.

Further Readings and Information Sources

Virtually every book written on software project management contains a discussion of scheduling. Wysoki (*Effective Project Management,* Wiley, 2006), Lewis (*Project Planning Scheduling and Control,* 4th ed., McGraw-Hill, 2006), Luckey and Phillips (*Software Project Management for Dummies,* For Dummies, 2006), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* 9th ed., Wiley, 2005), Hughes (*Software Project Management,* McGraw-Hill, 2005), The Project Management Institute (*PMBOK Guide,* 3d ed., PMI, 2004), Lewin (*Better Software Project Management,* Wiley, 2001), and Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3d ed., Wiley, 2000) contain worthwhile discussions of the subject. Although application specific, Harris (*Planning and Scheduling Using Microsoft Office Project 2007,* Eastwood Harris Pty Ltd., 2007) provides a useful discussion of how scheduling tools can be used to successfully track and control a software project.

Fleming and Koppelman (*Earned Value Project Management,* 3d ed., Project Management Institute Publications, 2006), Budd (*A Practical Guide to Earned Value Project Management,* Management Concepts, 2005), and Webb and Wake (*Using Earned Value: A Project Manager's Guide,* Ashgate Publishing, 2003) discuss the use of earned value techniques for project planning, tracking, and control in considerable detail.

A wide variety of information sources on software project scheduling is available on the Internet. An up-to-date list of World Wide Web references relevant to software project scheduling can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.