SOFTWARE ENGINEERING

UNIT IV

1.Software Testing Strategies: 2.A Strategic Approach to Software Testing – 3.Test Strategies for Conventional Software – 4.Unit Testing – 5.Integration Testing – 6.Validation Testing – 7.System Testing – 8.The Art of Debugging – 9.Testing Conventional Applications: 10.Software Testing Fundamentals – 11.White Box Testing – 12.Basis Path Testing – 13.Control Structure Testing – 14.Black-Box Testing – 15.Object-Oriented Testing Methods.

**1. Software Testing Strategies:**

Astrategy 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 only defence against programming errors was careful design and the native intelligence of the programmer. We are now in an era in which modern design techniques [and 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.

2.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.

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.

1. **Verification and Validation**
2. **Organizing for Software Testing**
3. **Software Testing Strategy—The Big Picture**
4. **Criteria for Completion of Testing**

**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:

**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.

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.”

**2. Organizing for Software Testing**

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

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

There are often a number of misconceptions that 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.

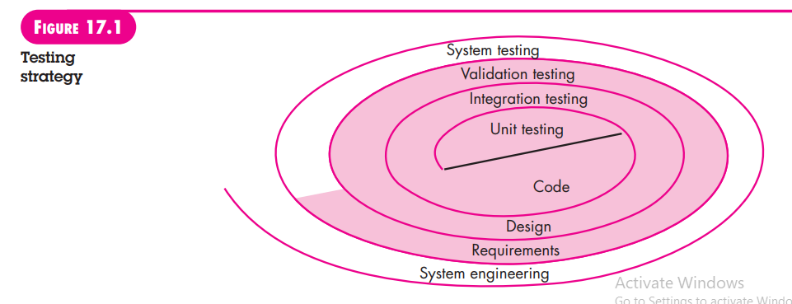
The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. After all, ITG personnel are paid to find errors.

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.

**3. Software Testing Strategy—The Big Picture**

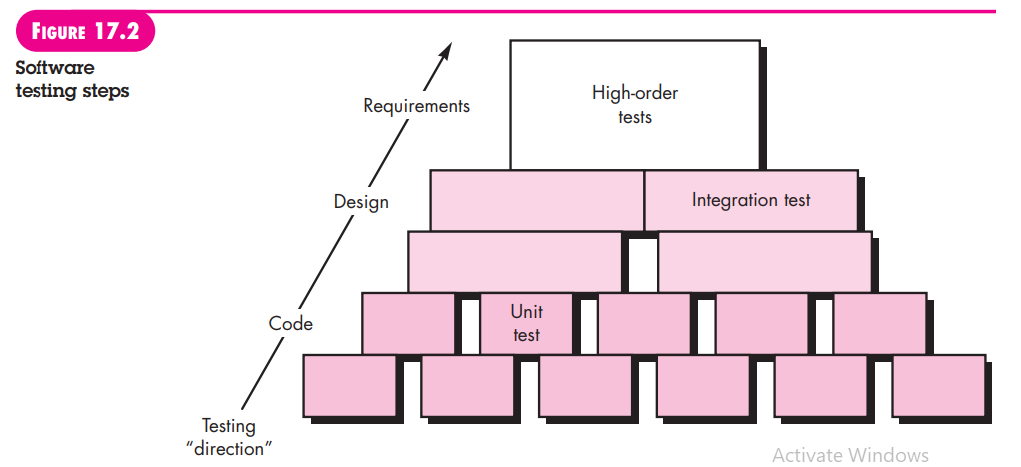
The software process may be viewed as the spiral illustrated in Figure 17.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. 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.

****

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.

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 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.



**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.

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.

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.

**3. Test Strategies for Conventional Software**

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.

**1. Unit Testing**

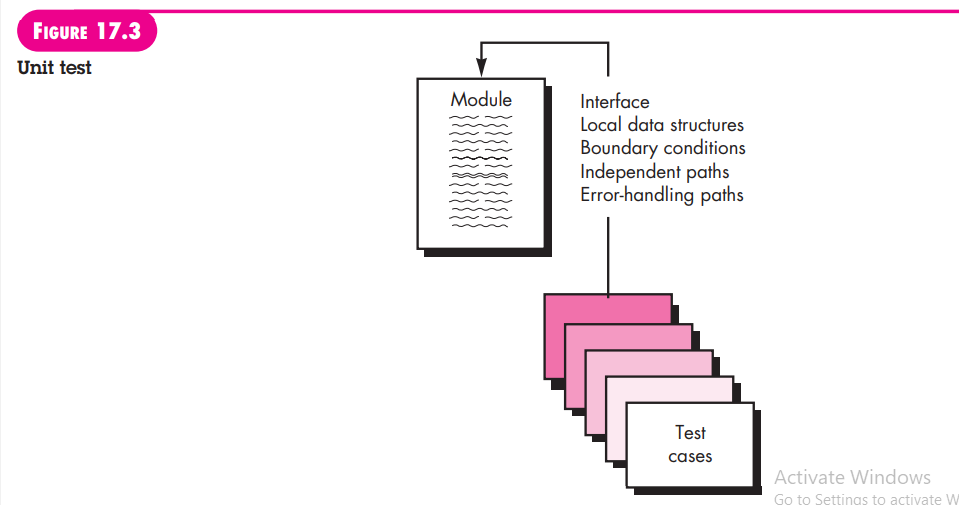
* 1. **Unit-test considerations.**
  2. **Unit-test procedures.**

**2. Integration Testing**

* 1. **Top-down integration**
  2. **Bottom-up integration.**
  3. **Regression testing.**
  4. **Smoke testing.**
  5. **Strategic options.**
  6. **Integration test work products**.

**1. Unit Testing**

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

****

1. **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.

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 nth element of an n-dimensional array is processed, when the ith 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.

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!

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,

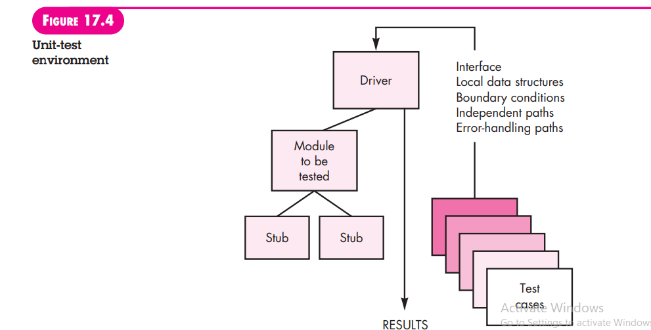
(4) exception-condition processing is incorrect, or

(5) error description does not provide enough information to assist in the location of the cause of the error.

**b. 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 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.



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.

**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.

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

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

1. **Top-down integration**

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

Referring to Figure 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 M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 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 M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows. The integration process is performed in a series of five steps:

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

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.

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.

1. **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:

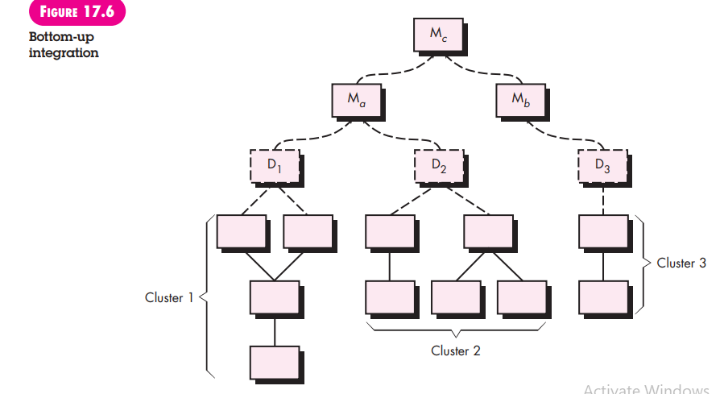
1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in Figure 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 Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.



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

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

• A representative sample of tests that will exercise all software functions.

• Additional tests that focus on software functions that are likely to be affected by the change.

• Tests that focus on the software components that have been changed.

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

1. **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 “showstopper” errors that have the highest likelihood of throwing the software project behind schedule.

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

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell [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:   
• 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.

1. **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.

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.

1. **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:

**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.

**4. Unit Testing**

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.

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.

**5. Integration Testing**

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].

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.

**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.

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.***

**1. Validation-Test Criteria**

**2. Configuration Review**

**3. Alpha and Beta Testing**

**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.

**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.

**3. Alpha and Beta Testing**

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

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

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

The alpha test is conducted at the developer’s site by a 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.

**7. System Testing**

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.

1. Recovery Testing

2. Security Testing

3. Stress Testing

4. Performance Testing

5. Deployment Testing

**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

**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.

**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.

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.

**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.

**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.

8. The Art of Debugging

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

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.

**1. The Debugging Process**

**2. Psychological Considerations**

**3. Debugging Strategies**

**a. Debugging tactics**

**b. Automated debugging**

**c. The people factor**

**4. Correcting the Error**

**1. The Debugging Process**

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 non corresponding 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? 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).

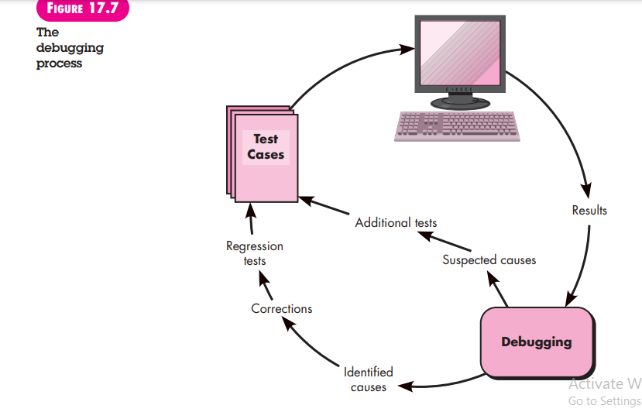
4. The symptom may be caused by human error that is not easily traced.

5. 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.

**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.

**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:

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

**a. 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.

**b. 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 languagespecific 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.

**c. 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!”

**4. Correcting the Error**

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck [Van89] suggests three simple questions that 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.

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.

9. Testing Conventional Applications:

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.

10. Software Testing Fundamentals

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.

**1. Testability**

**2. Test Characteristics**

**1. Testability**

James Bach1 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.

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

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

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

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

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

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

**2. Test Characteristics**.

And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of 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.

11. White Box Testing

White-box testing, sometimes called glass-box testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.

Using white-box testing methods, 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.

12.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.

**1. Flow Graph Notation**

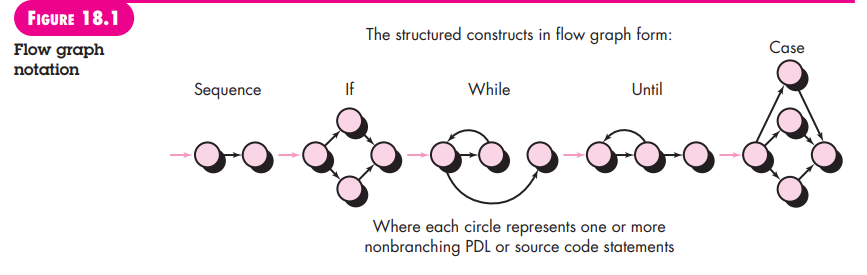
**2. Independent Program Paths**

**3. Deriving Test Cases**

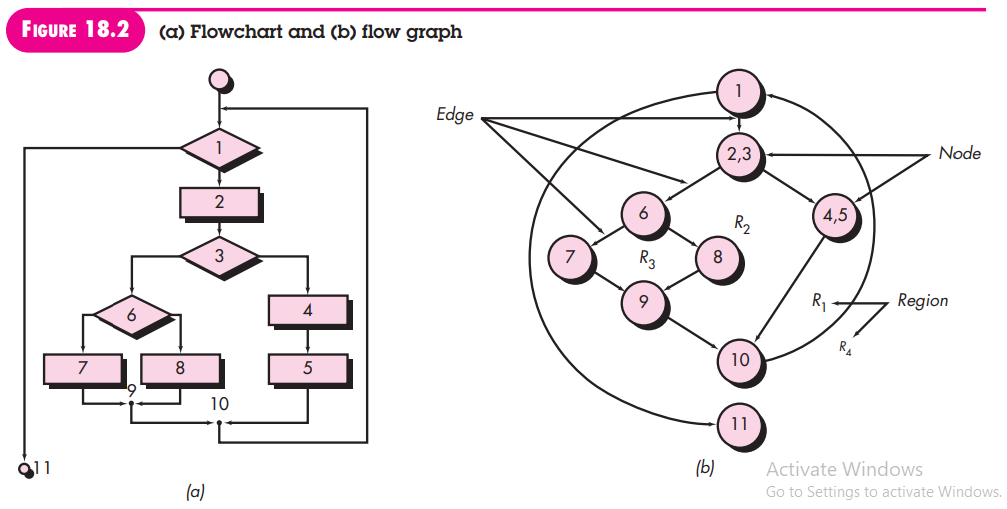
**4. Graph Matrices**

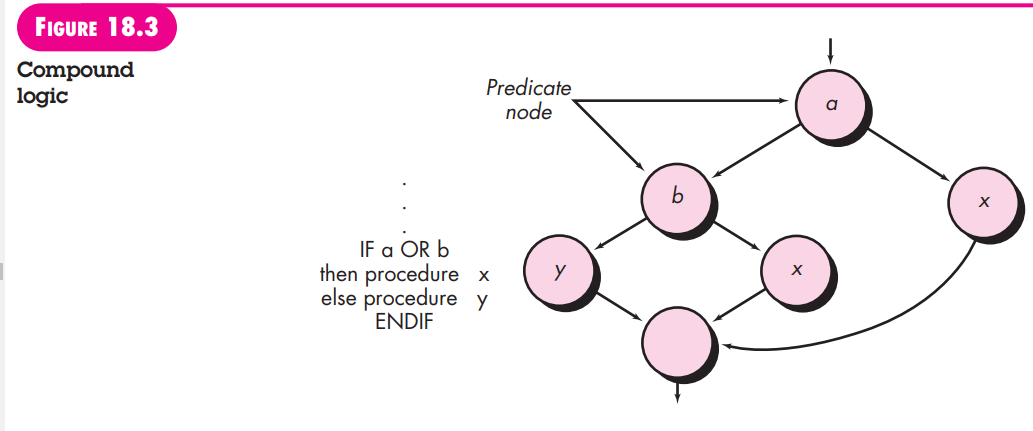
**1. Flow Graph Notation**

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.



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





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.

**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. 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:

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:

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.

**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:

***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

**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;

sum = 0;

DO WHILE value[i] <> –999 AND total.input < 100

increment total.input by 1;

ENDDO

IF total.valid > 0

ENDIF END average

IF value[i] > = minimum AND value[i] < = maximum

THEN average = sum / total.valid; ELSE

**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

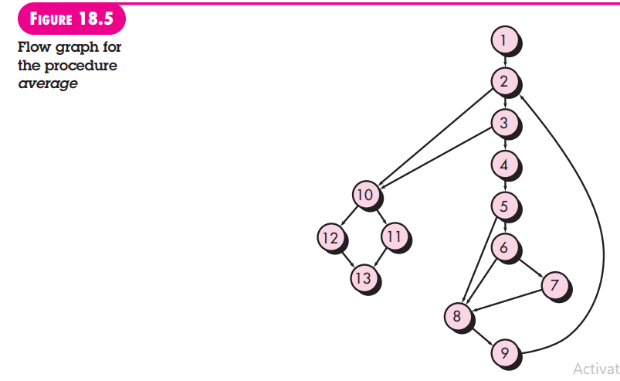
Path 2: 1-2-10-12-13

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.

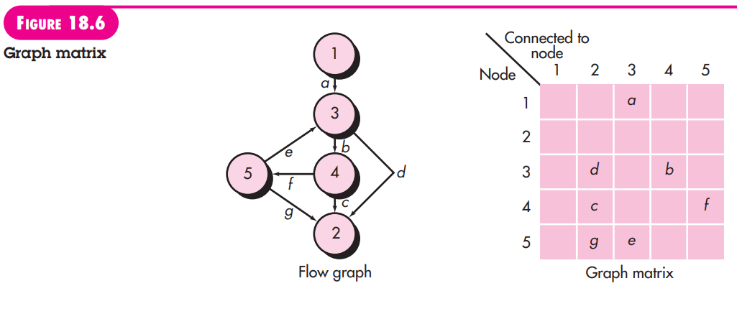
***4. Prepare test cases that will force execution of each path in the basis set.*** Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results. Once all test cases have been completed, the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

**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.



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. 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.

13. Control Structure Testing

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.

**1. Condition Testing**

**2. Data Flow Testing**

**3. Loop Testing**

**a. Simple loops.**

**b. Nested loops**

**c. Concatenated loops**

**d. Unstructured loops**

**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

E1 <relational operator>E2

where E1 and E2 are arithmetic expressions and <relational operator> is one of the following:

<,<=,=,=/(non-equality),>,or >=.

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(&) ,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.

**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,

DEF(S) {X | statement S contains a definition of X}

USE(S) {X | statement S contains a use of X}

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

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

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

**3. Loop Testing**

**a. Simple loops.**

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

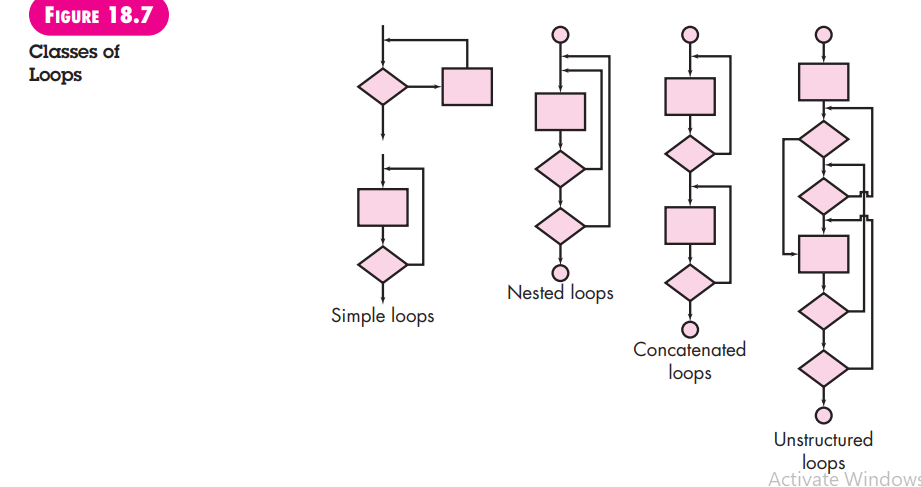
1. Skip the loop entirely.

2. Only one pass through the loop.

3. Two passes through the loop.

4. m passes through the loop where m < n.

5. n - 1, n, n + 1 passes through the loop.

****

**b. 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.

**c. 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.

**d. Unstructured loops**

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

14. 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 whitebox 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, blackbox 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:

• 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.

1. Graph-Based Testing Methods

2. Equivalence Partitioning

3. Boundary Value Analysis

4. Orthogonal Array Testing

15. Object-Oriented Testing Methods

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

However, an overall approach to OO test-case design has been suggested by Berard [Ber93]:

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.

2. The purpose of the test should be stated.

3. A list of testing steps should be developed for each test and should contain:

a. A list of specified states for the class that is to be tested

b. A list of messages and operations that will be exercised as a consequence of the test

c. A list of exceptions that may occur as the class is tested

d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)

e. Supplementary information that will aid in understanding or implementing the test

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

**1. The Test-Case Design Implications of OO Concepts**

**2. Applicability of Conventional Test-Case Design Methods**

**3. Fault-Based Testing**

**4. Test Cases and the Class Hierarchy**

**5. Scenario-Based Test Design**

**6. Testing Surface Structure and Deep Structure**

**1. The Test-Case Design Implications of OO Concepts**

As a class evolves through the requirements and design models, it becomes a target for test-case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing. As Binder [Bin94a] notes, “Testing requires reporting on the concrete and abstract state of an object.” Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

Inheritance may also present you with additional challenges during test-case design. I have already noted that each new usage context requires retesting, even though reuse has been achieved. In addition, multiple inheritance4 complicates testing further by increasing the number of contexts for which testing is required [Bin94a]. If subclasses instantiated from a superclass are used within the same problem domain, it is likely that the set of test cases derived for the superclass can be used when testing the subclass. However, if the subclass is used in an entirely different context, the superclass test cases will have little applicability and a new set of tests must be designed.

**2. Applicability of Conventional Test-Case Design Methods**

The white-box testing methods described in Chapter 18 can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested. However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods. As I noted in Chapter 18, use cases can provide useful input in the design of black-box and state based tests.

**3. Fault-Based Testing**

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

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

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

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

**4. Test Cases and the Class Hierarchy**

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

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

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

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

**5. Scenario-Based Test Design**

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

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

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

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

**Use Case: Fix the Final Draft**

**Background:**

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

1. Print the entire document.

2. Move around in the document, changing certain pages.

3. As each page is changed, it’s printed.

4. Sometimes a series of pages is printed.

This scenario describes two things: a test and specific user needs. The user needs are obvious:

(1) a method for printing single pages and

(2) a method for printing a range of pages. As far as testing goes, there is a need to test editing after printing (as well as the reverse). Therefore, you work to design tests that will uncover errors in the editing function that were caused by the printing function; that is, errors that will indicate that the two software functions are not properly independent.

**Use Case: Print a New Copy**

**Background:**

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

1. Open the document.

2. Print it.

3. Close the document.

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

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

**Use Case: Print a New Copy**

1. Open the document.

2. Select “Print” in the menu.

3. Check if you’re printing a page range; if so, click to print the entire document.

4. Click on the Print button.

5. Close the document. But this scenario indicates a potential specification error.

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

You might miss this dependency as you design tests, but it is likely that the problem would surface during testing. You would then have to contend with the probable response, “That’s the way it’s supposed to work!”

**6. Testing Surface Structure and Deep Structure**

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

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

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

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

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