UNIT – IV

Testing Strategies:

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

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses. Shooman [Sho83] discusses these issues:

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

These “approaches and philosophies” are what we call strategy—here the testing methods and techniques that implement the strategy are presented.

A strategic approach to software testing:

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

- A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:
- To perform effective testing, you should conduct effective technical re- views (Chapter 20). 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

Software testing is one element of a broader topic that is often referred to as ver- ification and validation (V&V). Verification refers to the set of tasks that ensure that software correctly implements a specific function. Validation refers to a dif- ferent 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

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 mitigates 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 to uncover errors. Unfortunately, errors will be nevertheless present. And, if the software engineer doesn't find them, the customer will!

There are often a number of misconceptions that you might infer 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 team

3. Software Testing Strategy

The software process may be viewed as the spiral illustrated in the below Figure. 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 along streamlines that decrease the level of abstraction on each turn.

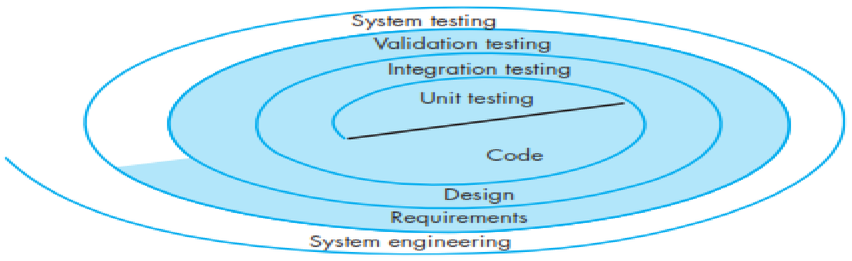
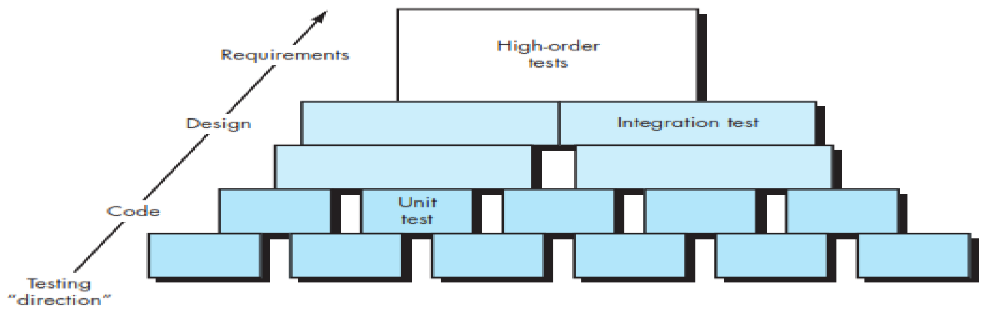


Fig: Testing Strategy

A strategy for software testing may also be viewed in the context of the spiral (Above Figure). 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 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 the below Figure. 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 pro- gram construction. Test-case design techniques that focus on inputs and out- puts 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 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 exe- cutes 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 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. By collecting metrics during software testing and making use of existing statistical models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing".

Test strategies for conventional software"

Many strategies 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 the hope 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.

A testing strategy that is chosen by many 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 (sometimes on a daily basis), 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

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

Unit Test Considerations. Unit tests are illustrated schematically in the below Figure. 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.

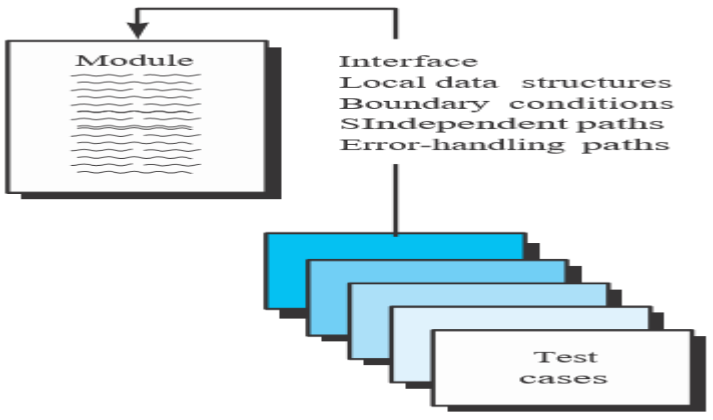


Fig: UNIT TEST

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

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. Your- don [You75] calls this approach anti bugging. Unfortunately, there is a tendency to incorporate error handling into software and then never test the error handling. If error-handling paths are implemented, they must be tested.

Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

Unit-Test Procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results. Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. The unit test environment is illustrated in the below Figure. 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 coded (formal design is not commonly applied) but that is not delivered with the final software

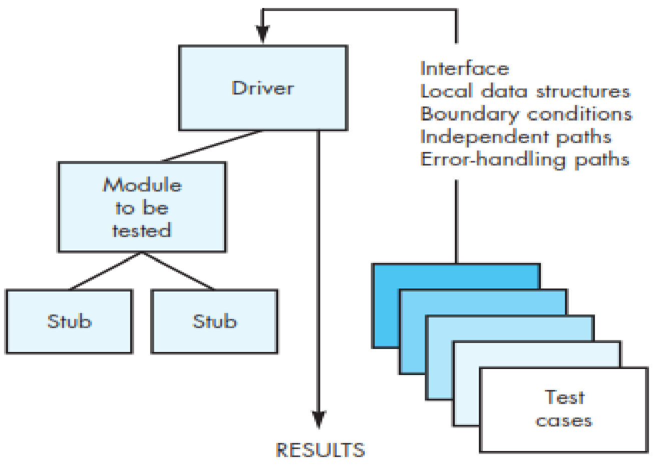


FIG: Unit-test environment

product. If drivers and stubs are kept simple, actual over- head 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).

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 "put- ting them together"—interfacing. Data can be lost across an interface; one component can have an inadvertent, adverse effect on another; sub functions, 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 pro- gram structure that has been dictated by design.

There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance and the entire program is tested as a whole. Chaos usually results! Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.

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

Top-Down Integration. Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (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.

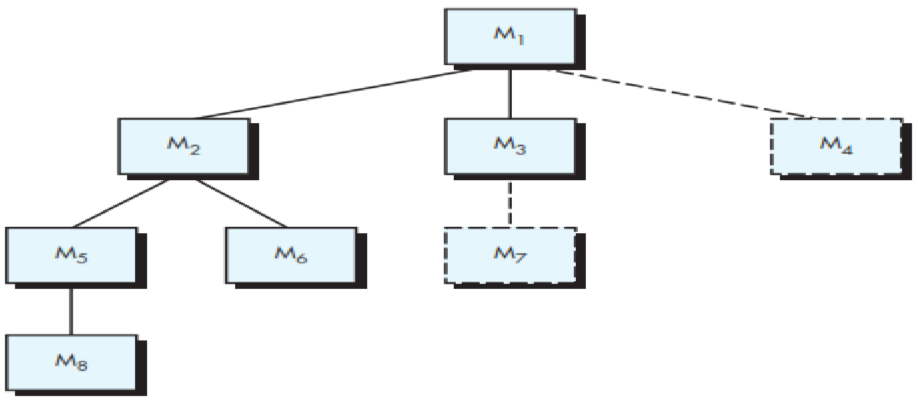


Fig: Top Down Integration

Referring to above Figure, depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is some- what 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.

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 sub function
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.

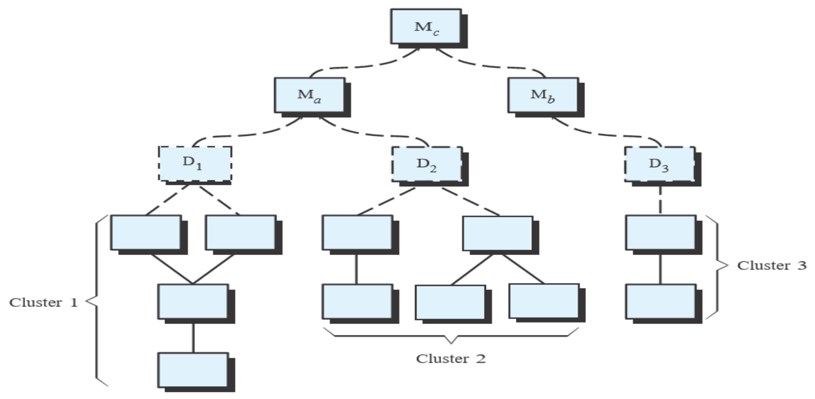


Fig: Bottom-Up Integration

Integration follows the pattern illustrated in the above Figure. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M. Drivers D1 and D2 are removed and the clusters are interfaced directly to M. 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.

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 different software categories 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

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). If a deviation from specification is uncovered, a deficiency list is created. A method for resolving deficiencies (acceptable to stakeholders) must be established.

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.

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 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.

System testing:

At the beginning, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall out- side 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.

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), re initialization, check pointing 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 sys- tem 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."

3. Stress Testing

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

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate 10 interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other re- sources 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.

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.

The Debugging Process:

Debugging is not testing but often occurs as a consequence of testing. Referring to figure, , the debugging process begins with the execution of a test case.

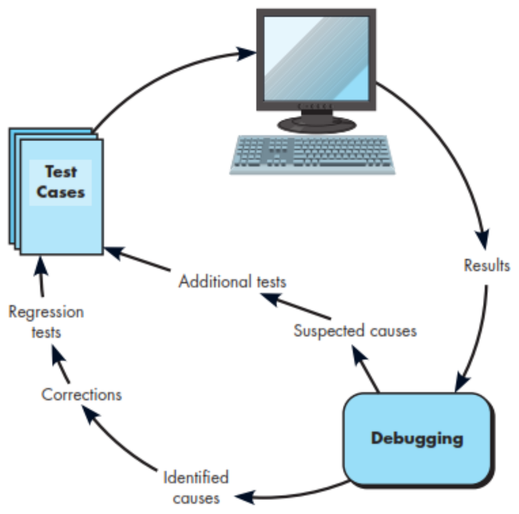


Fig: The Debugging Process

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.

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 exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

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

Testing Conventional Applications:

Testing presents an interesting dilemma 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!"

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.

Testability.

James Bach1 provides the following definition for testability: "Soft- ware 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 test- ing 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 software work products that are amenable to testing.

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.

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 dictate the execution of only those tests that has the highest likelihood of uncovering a whole class of errors.

A good test should be neither too simple nor too complex. Although it is some- times 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.

White-Box testing:

White-box testing, sometimes called glass-box testing or structural 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.

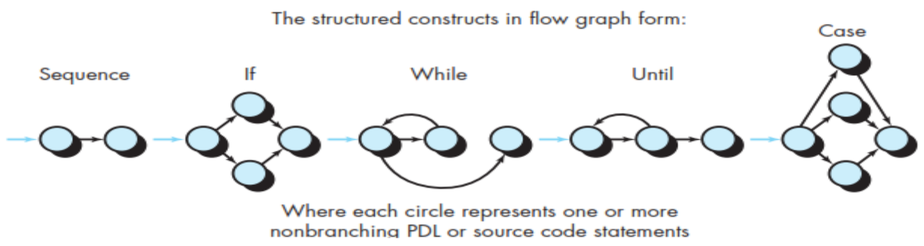
Basis path testing:

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 guar- anteed to execute every statement in the program at least one time during testing.

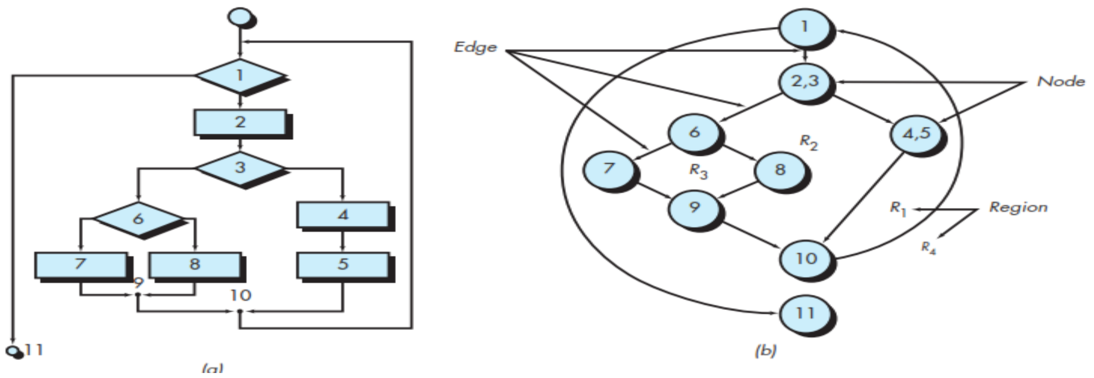
Flow Graph Notation

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

Flow graph notation


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

(a) Flowchart and (b) flow graph



To illustrate the use of a flow graph, consider the procedural design representation in the Figure.a. Here, a flowchart is used to depict program control structure. In the above Figure.b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to above Figure.b, 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.

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 below Figure, 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.

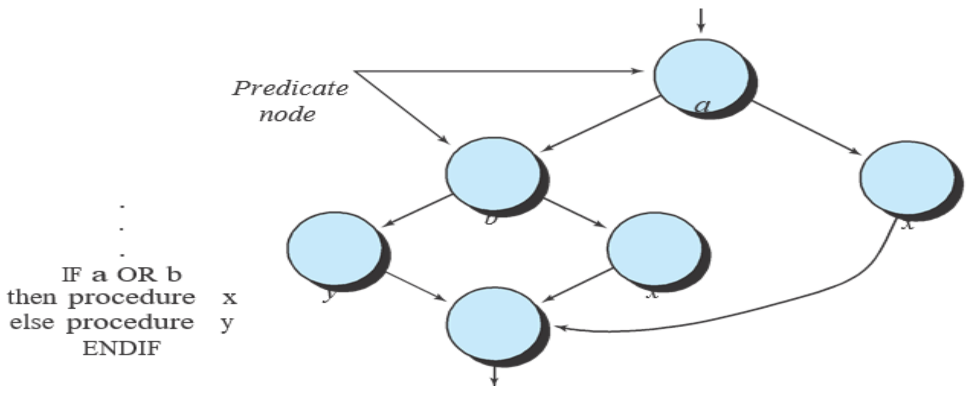


Fig. Compound Logic

1. 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 the above flow graph Figure b is

Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-
Path 3: 1-2-3-6-8-9-10-
Path 4: 1-2-3-6-7-9-10-

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 the above Figure.b. 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) 5 E 2 N 1 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) 5 P 1 1

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

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

1. The flow graph has four regions
2. V(G) 5 11 edges 2 9 nodes 1 2 5 4
3. V(G) 5 3 predicate nodes 1 1 5 4
4. Therefore, the cyclomatic complexity of the flow graph in the above flow graph Figure b is 4.

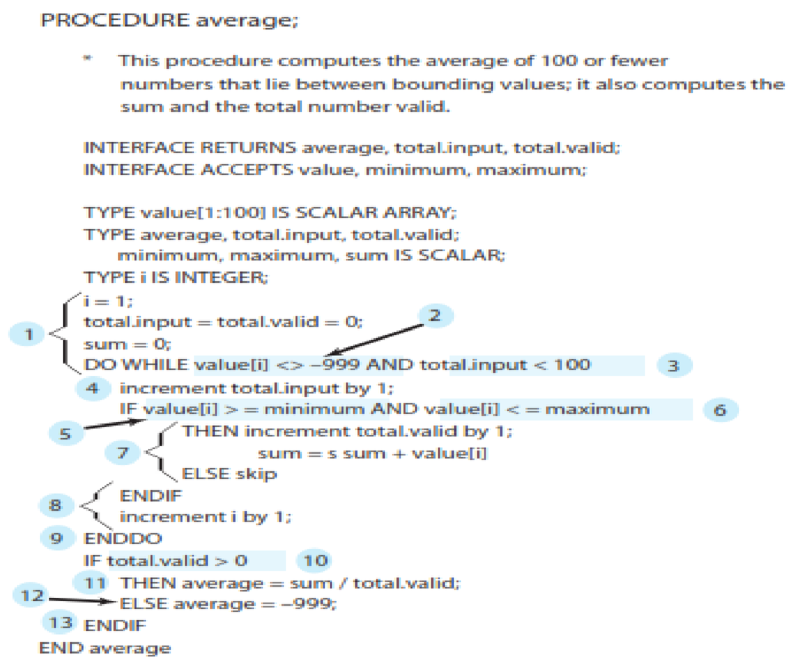
   More important, the value for V(G) provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

2. Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure average, depicted in PDL in Figure 23.4, 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 23.4.1. Referring to the PDL for average in the below Figure 23.4, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in the below flow graph Figure.



```
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;
        IF value[i] > = minimum AND value[i] < = maximum
            THEN increment total.valid by 1;
                    sum = s sum + value[i]
            ELSE skip
        ENDIF
        increment i by 1;
    ENDDO
    IF total.valid > 0
        THEN average = sum / total.valid;
        ELSE average = -999;
    ENDIF
END average
```

Fig: The PDL with nodes identified.

The corresponding flow graph is in the below Figure.

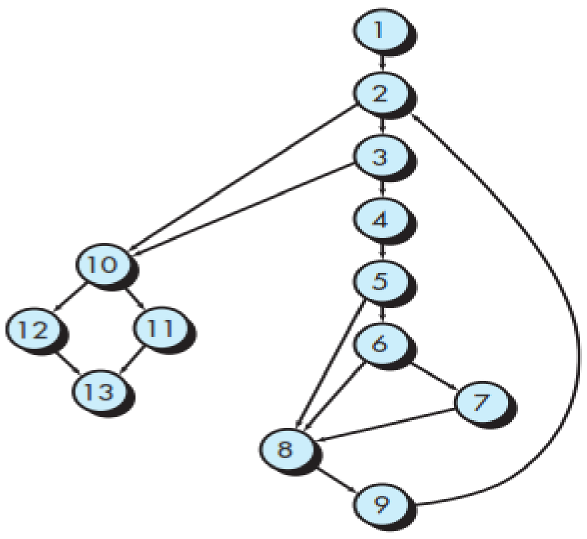


Fig: Flow graph for the Procedure Average

2. Determine the cyclomatic complexity of the resultant flow graph. The cyclomatic complexity V (G) is determined by applying the algorithms described in Section 23.4.2. 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 the above Figure.

V (G) 5 6 regions

V (G) 5 17 edges 2 13 nodes 1 2 5 6

V (G) 5 5 predicate nodes 1 1 5 6

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

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

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

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

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

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

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

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

4. Prepare test cases that will force execution of each path in the basis set.

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

It is important to note that some independent paths (e.g., path 1 in our ex- ample) 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 pro- gram. In such cases, these paths are tested as part of another path test.

Control structure testing:

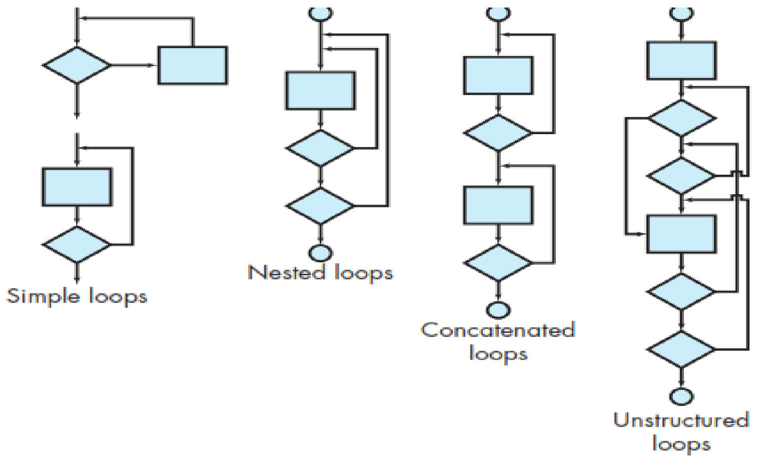
The basis path testing technique described in Section 23.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module.

Data flow testing [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.

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

Classes of Loops



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

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

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

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

Concatenated Loops. Concatenated loops can be tested using the approach de- fined 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 in- dependent, the approach applied to nested loops is recommended.

Unstructured Loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

Black-Box testing:

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

Black-box testing attempts to find errors in the following categories:

(1) Incorrect or missing functions,

(2) Interface errors,

(3) Errors in data structures or external database access,

(4) Behavior or performance errors, and

(5) Initialization and termination errors.

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

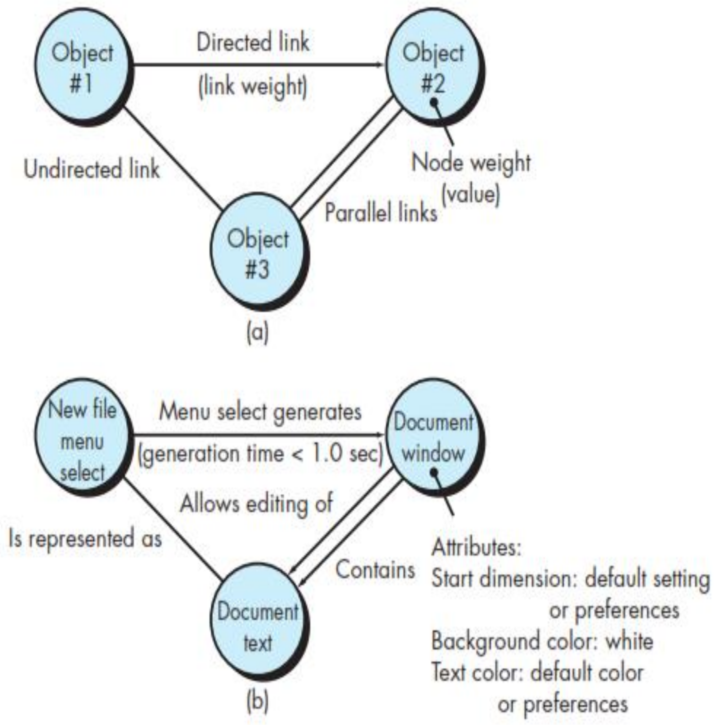
- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system 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 fol- lowing criteria [Mye79]: 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 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.

Graph-Based Testing Methods:

The first step in black-box testing is to understand the objects5 that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify “all objects have the expected relationship to one another” [Bei95]. Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

(a) Graph notation; (b) simple example



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

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

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

Object #1 5 **newFile (**menu selection)

Object #2 5 **documentWindow**

Object #3 5 **documentText**

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

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps. For example, a data object **flightInformationInput** is followed by the operation **validationAvailabilityProcessing().**

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

Data flow modeling.The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node **FICATaxWithheld (FTW)** is computed from gross wages **(GW)** using the relationship, **FTW 5 0.62 3 GW**.

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

Equivalence Partitioning:

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

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

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

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

Boundary Value Analysis:

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

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

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

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

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