UNIT – 5

Unit V:

Software Testing Strategies: A strategic Approach to Software Testing, Strategic Issues, Test Strategies for Conventional Software, Test Strategies for Object-Oriented Software, Test Strategies for WebApps, Validation Testing, System Testing, The Art of Debugging.

Testing Conventional Applications: Software Testing Fundamentals, Internal and External Views of Testing, White-Box Testing, basic Path testing, Control Structure Testing, Black-Box Testing, Model-based Testing, Testing for Specialized Environments, Architectures and Applications, Patterns for Software Testing.Testing Object-Oriented Applications: Broadening the View of Testing, Testing with OOA and OOD Models, Object-Oriented Testing Strategies, Object-Oriented Testing Methods, Testing Methods Applicable at the Class level, Interclass Test-Case Design.

5.0 SOFTWARE TESTING

What is it? Software is tested to uncover errors that were made inadvertently as it was designed and constructed.

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

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

What are the steps? Testing begins “in the small” and progresses “to the large.” By this I mean that early testing focuses on a single component or a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements. As errors are uncovered, they must be diagnosed and corrected using a process that is called debugging.

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

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

5.1 A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. A number of software testing strategies have been proposed. 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. 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 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.

5.1.1 Verification and Validation: Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of tasks that ensure that software correctly implements a specific function. Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm 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

5.1.2 Organizing for Software Testing: 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?

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.

5.1.3 Software Testing Strategy—The Big Picture: The software process may be viewed as the spiral illustrated in Figure 17.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward 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.

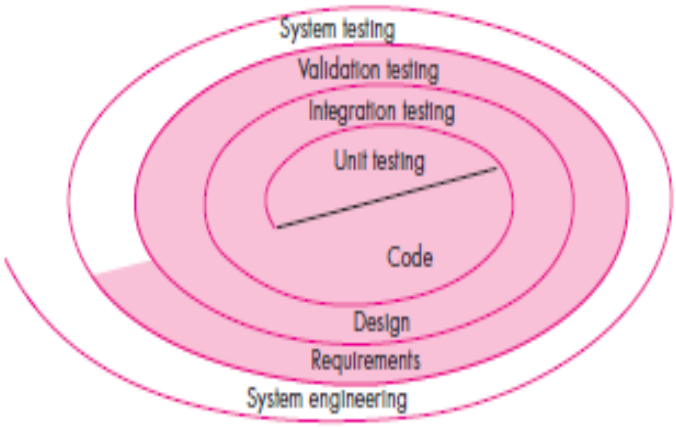

Fig 17.1: Testing strategy

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.*

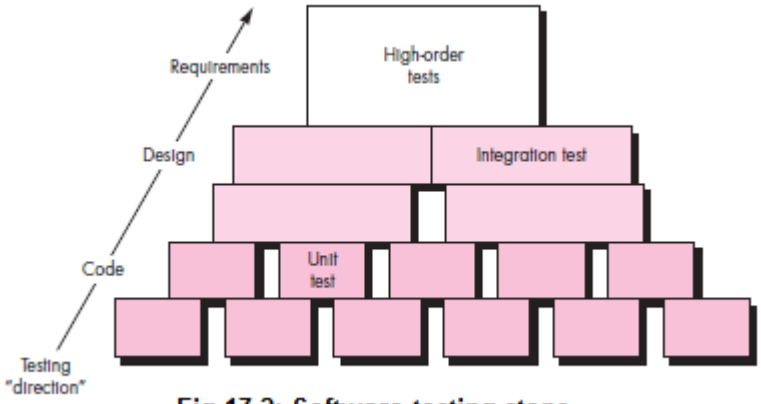

Fig 17.2: Software testing steps

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.
System testing: 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.

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

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.

Another response 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 *clean room software engineering* approach suggests statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population. Others 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 testing. The empirical approaches that currently exist are considerably better than raw intuition.

5.2 STRATEGIC ISSUES

Some times the best strategy will fail if a series of overriding issues are not addressed. Tom Gilb argues that a software testing strategy will succeed when software testers: Specify product requirements in a *quantifiable manner* long before testing commences.
Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as *portability*, *maintainability*, and *usability*. These should be specified in a way that is measurable so that testing results are unambiguous.

The specific objectives of testing should be stated in measurable terms. For example, *test effectiveness, test coverage, meantime-to-failure, the cost to find and fix defects, remaining defect density or frequency of occurrence*, and *test work-hours* should be stated in the *test plan*.

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

Develop a testing plan that emphasizes "rapid cycle testing." Gilb recommends that a software team "learn to test in rapid cycles of customer-useful, at least field 'trialable,' increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.
Build "robust" software that is designed to test itself. Software should be designed in a manner that uses anti bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
Use effective technical reviews as a filter prior to testing. Technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high quality software.
Conduct technical reviews to assess the test strategy and test cases themselves.Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

5.3 TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

There are many strategies that can be used to test software. 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.

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

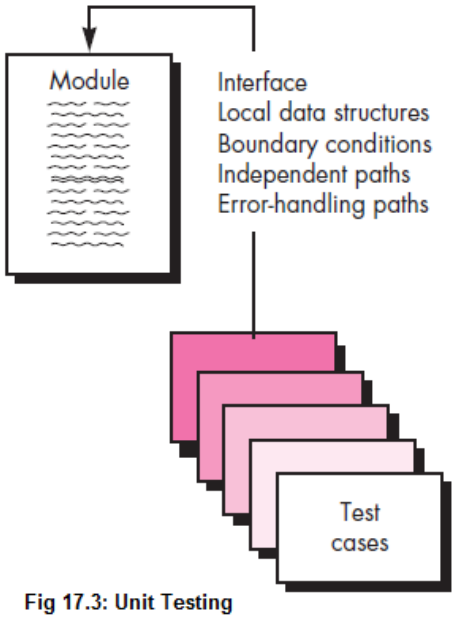

Fig 17.3: Unit Testing

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.

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

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the n^{th} element of an *n-dimensional* array is processed, when the 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.

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. 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.

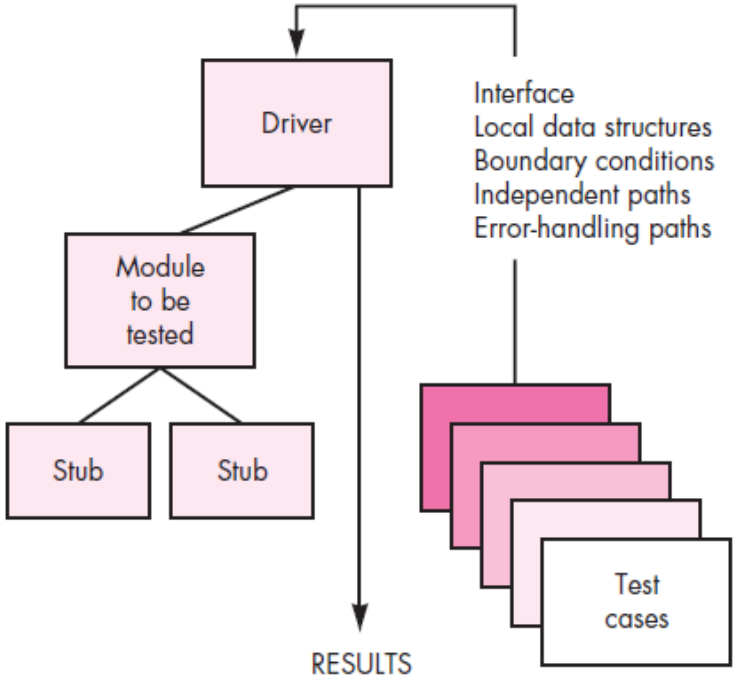

Fig 17.4: Unit-test environment

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.

5.3.2 Integration Testing: 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 non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. *The entire program is tested as a whole*. 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.

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).

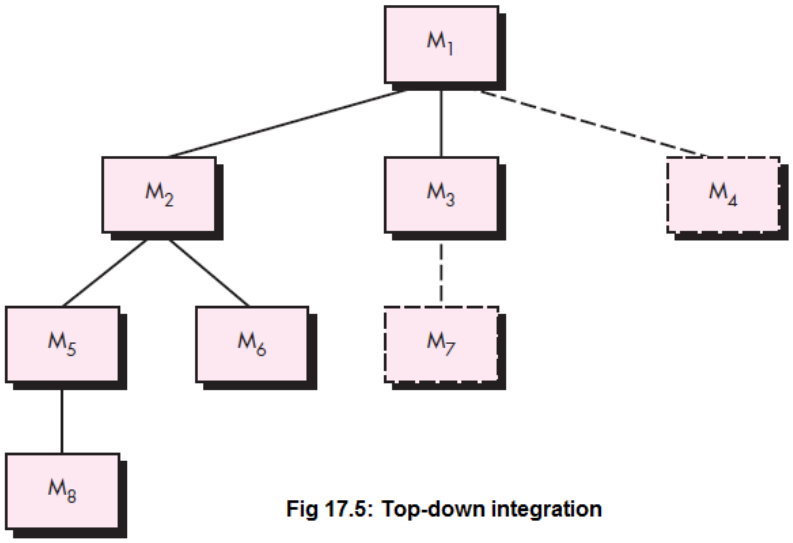

Fig 17.5: Top-down integration

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 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. 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.

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.

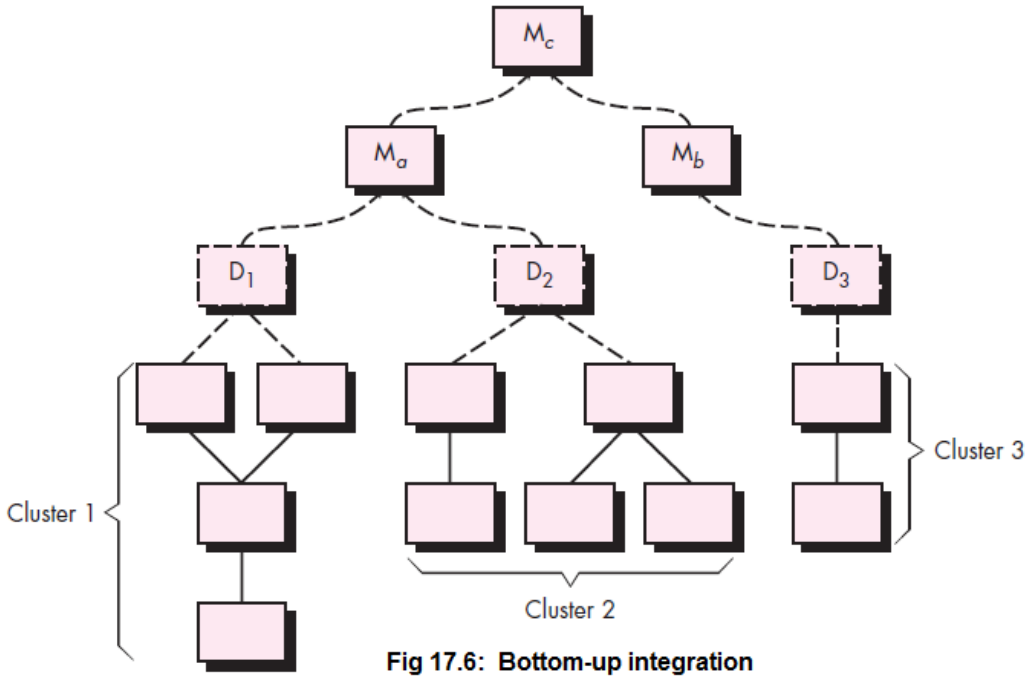

Fig 17.6: Bottom-up integration

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.*

Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools. 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.

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.

Smoke testing provides a number of benefits when it is applied on complex, timecritical, 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.

Strategic options. There has been much discussion 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*. 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.

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. 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. 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 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. 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.

5.4 TEST STRATEGIES FOR OBJECT-ORIENTED SOFTWARE

Although this fundamental objective remains unchanged for object-oriented software, the nature of object-oriented software changes both testing strategy and testing tactics.

5.4.1 Unit Testing in the OO Context: When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the

operations that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

You can no longer test a single operation in isolation 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 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.

5.4.2 Integration Testing in the OO Context: Because object-oriented software does not have an obvious hierarchical control structure, traditional top-down and bottom-up integration strategies have little meaning.
There are two different strategies for integration testing of OO systems. 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.

5.5 TEST STRATEGIES FOR WEBAPPS

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

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

The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

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

5.6 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.

5.6.1 Validation-Test Criteria: Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all

functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).
After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created. Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

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

5.6.3 Alpha and Beta Testing: It is virtually impossible for a software developer to foresee how the customer will really use a program. 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.

5.7 SYSTEM TESTING

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.

5.7.1 Recovery Testing: *Many computer-based systems must recover from faults and resume processing with little or no downtime*. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.
Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the *mean-time-to-repair* (MTTR) is evaluated to determine whether it is within acceptable limits.

5.7.2 Security Testing: Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: *hackers* who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.
Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

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.

5.7.3 Stress Testing: 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.

5.7.4 Performance Testing: 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.

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

5.8 THE ART OF DEBUGGING

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.

5.8.1 The Debugging Process: Debugging is not testing but often occurs as a consequence of testing. 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 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 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.

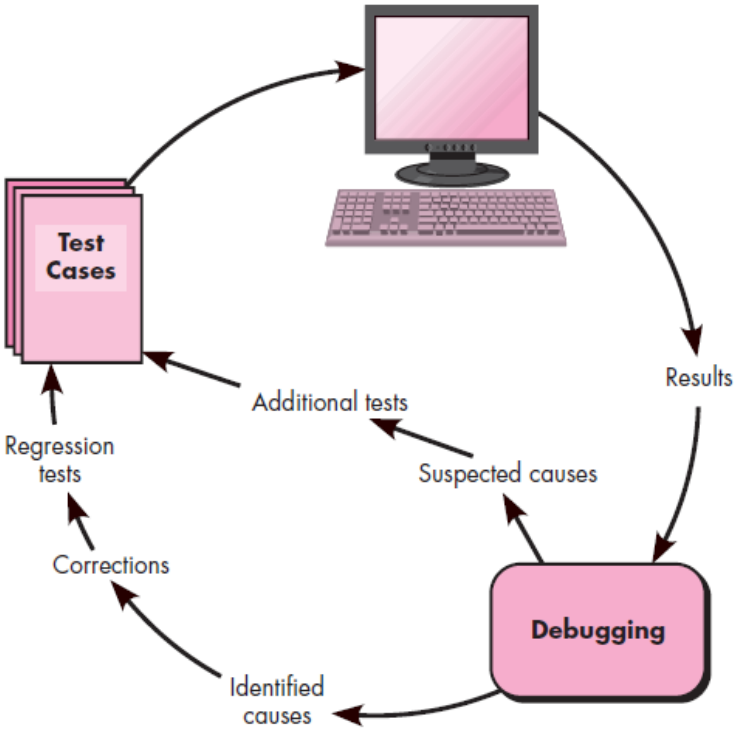

Fig 17.7: The debugging process

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.

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

5.8.3 Debugging Strategies: In general, three debugging strategies have been proposed *(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.
Debugging tactics. The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements. You hope that somewhere in the morass of information that is produced you'll find a clue that can lead to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.
The third approach to debugging—*cause elimination*—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Automated debugging. Each of these debugging approaches can be supplemented with debugging tools that can provide you with semi automated support as debugging strategies are

attempted. Integrated development environments (IDEs) provide a way to capture some of the language specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation." A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

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. A final maxim for debugging might be: "When all else fails, get help!"

5.8.4 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 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. 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.

5.9 Testing Conventional Applications

What is it? Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic and interfaces of every software component and (2)exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.
Who does it? During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved.
Why is it important? Reviews and other SQA actions can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.
What are the steps? For conventional applications, software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test-case design techniques and (2) software requirements are exercised using "black box" test-case design techniques. Use cases assist in the design of tests to uncover errors at the software validation level. In every case, the intent is to find the maximum number of errors with the minimum amount of effort and time.
What is the work product? A set of test cases designed to exercise both internal logic, interfaces, component collaborations, and external requirements is designed and documented, expected results are defined, and actual results are recorded.
How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness. In addition, you can evaluate test coverage and track error detection activities.

5.9.1 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. The tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.
Testability. James Bach 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.

Test Characteristics : *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". 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.

5.9.2 INTERNAL AND EXTERNAL VIEWS OF TESTING: Any engineered product (and most other things) can be tested in one of two ways:
(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.
(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

The first test approach takes an external view and is called *black-box testing*. The second requires an internal view and is termed *white-box testing*.

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

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

5.9.3 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.

5.9.4 BASIS PATH TESTING: Basis path testing is a white-box testing technique first proposed by Tom McCabe. 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.

5.9.4.1 Flow Graph Notation: Before we consider the basis path method, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 18.1. Each structured construct has a corresponding flow graph symbol.

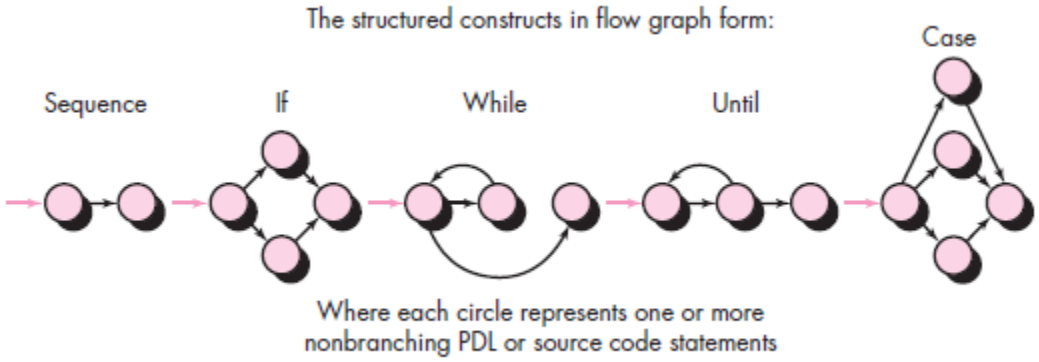

Fig 18.1: Flow graph notation

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 controland 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

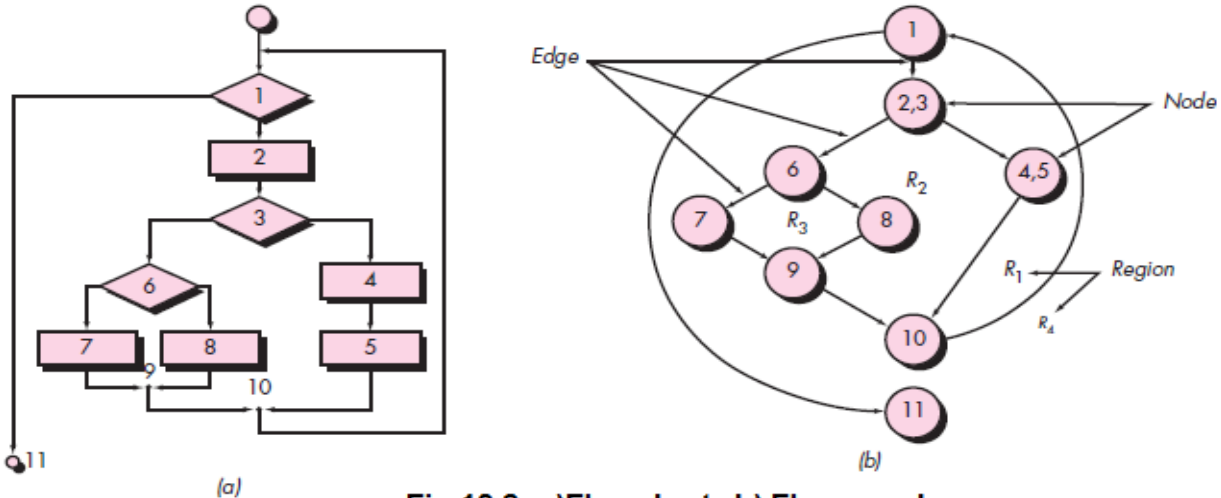

Fig 18.2: a)Flowchart b) Flow graph

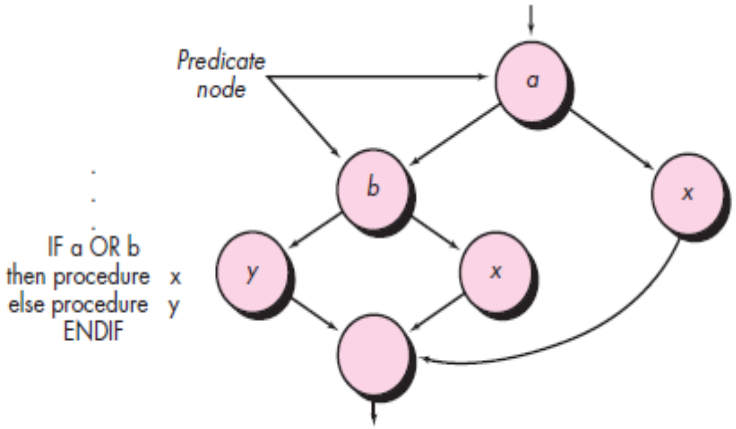

Fig 18.3: Compound logic

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

5.9.4.2 Independent Program Paths: An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed befor 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.

5.9.4.3 Deriving Test Cases: The basis path testing method can be applied to a procedural design or to source code. 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. 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

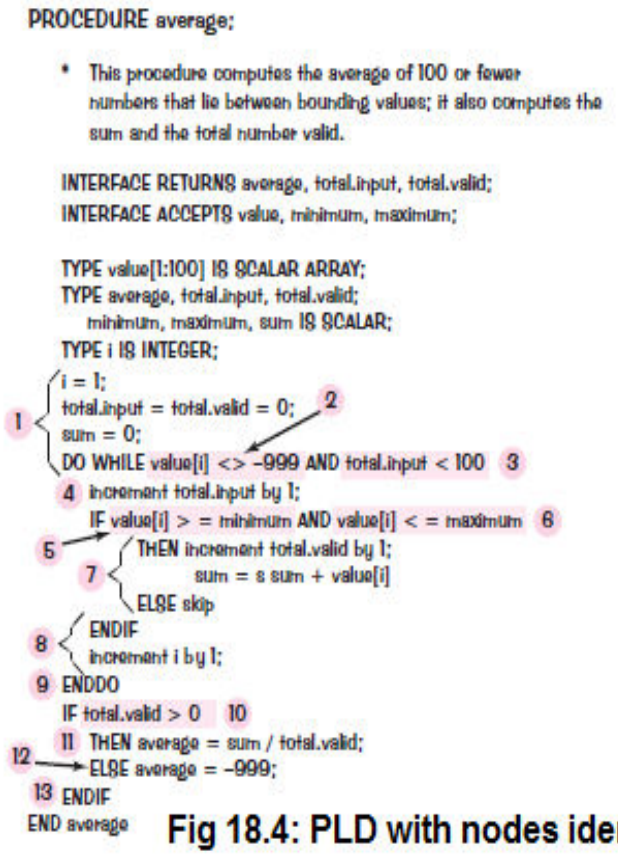

Fig 18.4: PLD with nodes identified

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

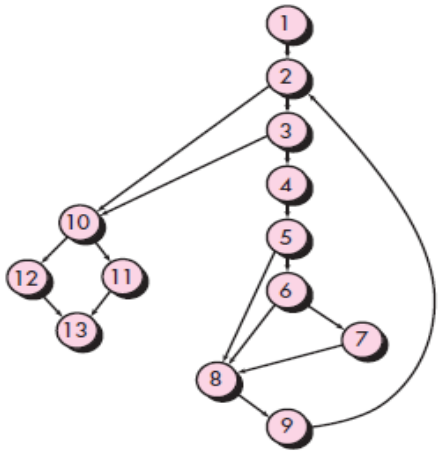

Fig 18.5: Flow graph for the procedure average

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.

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

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

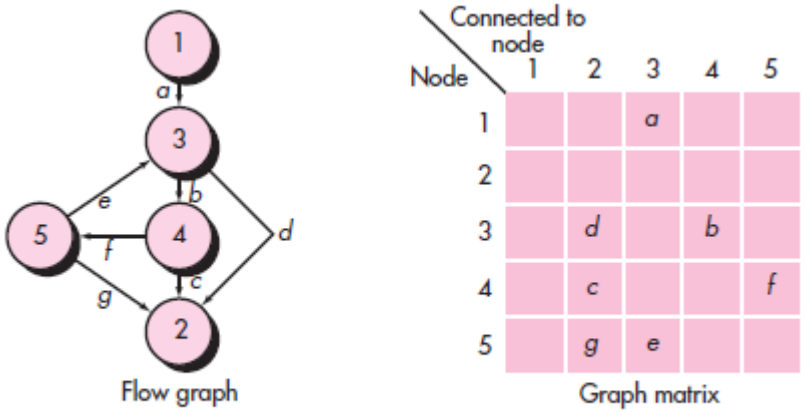

Fig 18.6: Graph Matrix

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

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 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.

5.9.5 CONTROL STRUCTURE TESTING

The basis path testing technique 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. The other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of *white-box testing*.

5.9.5.1 Condition Testing: Condition testing 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: <,>,>=,<=,!= (nonequality). A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.
If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and

arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

5.9.5.2 Data Flow Testing: The data flow testing method 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 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.

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

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

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

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

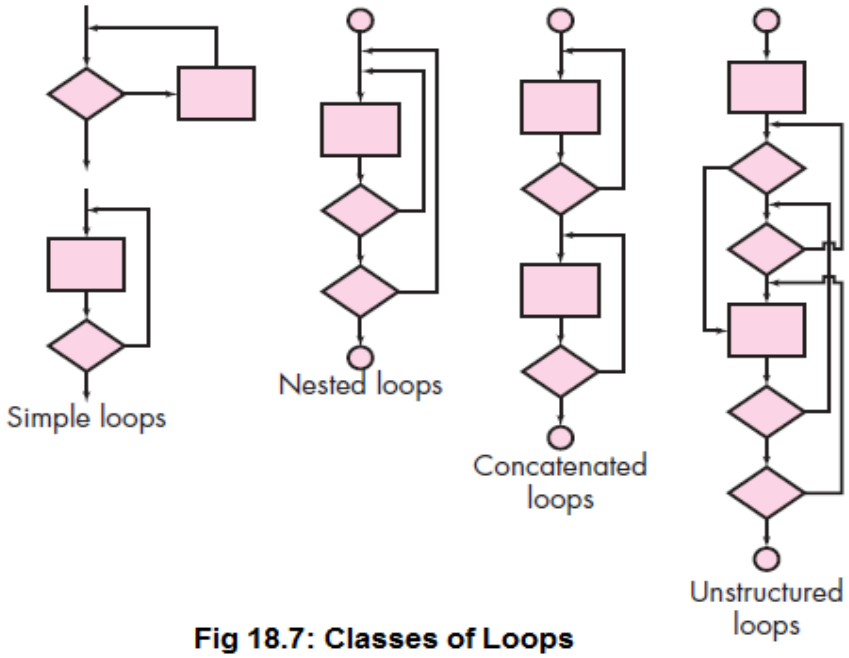

Fig 18.7: Classes of Loops

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

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

5.9.6 BLACK-BOX TESTING

Black-box testing, also called *behavioral testing*, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an

alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than 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. 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
(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.

5.9.6.1 Graph-Based Testing Methods: The first step in black-box testing is to understand the objects 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”. Stated in another way, software testing begins by creating a graph of important objects and their relationships and

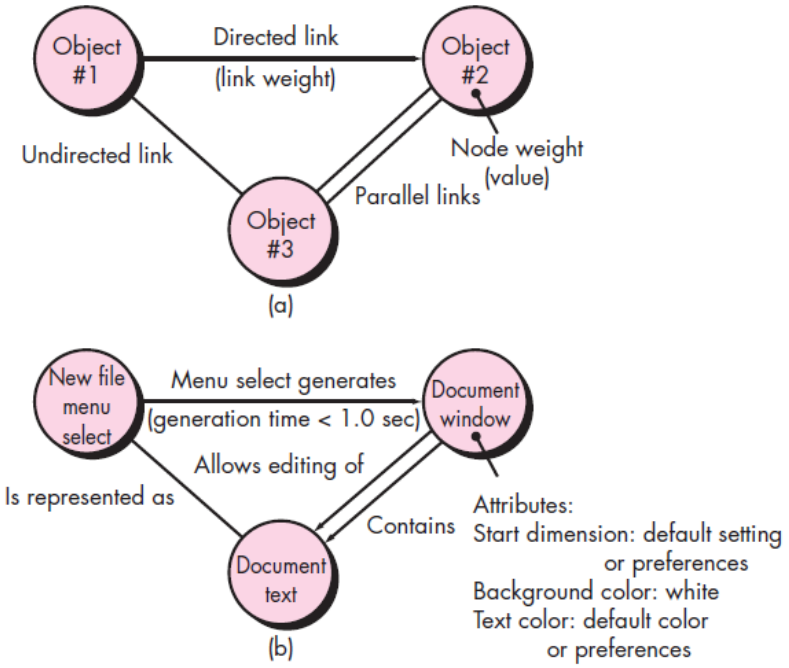

Fig 18.8: a)Graph notation b) Simple Example

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

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

The symbolic representation of a graph is shown in Figure 18.8a. Nodes are represented as circles connected by links that take a number of different forms.

A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link*, also called a *symmetric link*, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.
As a simple example, consider a portion of a graph for a word-processing application (Figure 18.8b) where
Object #1 _ **newFile** (menu selection)
Object #2 _ **documentWindow**
Object #3 _ **documentText**
Referring to the figure, a menu select on **newFile** generates a document window. The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated. The *link weight* indicates that the window must be generated in

less than 1.0 second. An *undirected link* establishes a symmetric relationship between the **newFile** menu selection and **documentText**, and parallel links indicate relationships between **documentWindow** and **documentText**.

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

Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., orderInformation is verified during **inventoryAvailabilityLook-up** and is followed by **customerBillingInformation input**). The state diagram can be used to assist in creating graphs of this type.
Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node FICA tax withheld (FTW) is computed from gross wages (GW) using the relationship.
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.

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

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

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

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.

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.

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

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

To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases. Phadke suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 18.9.

Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

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

To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function.
Each takes on three discrete values. For example, P1 takes on values:
P1 =1, send it now
P1 = 2, send it one hour later
P1 = 3, send it after midnight
P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.

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

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

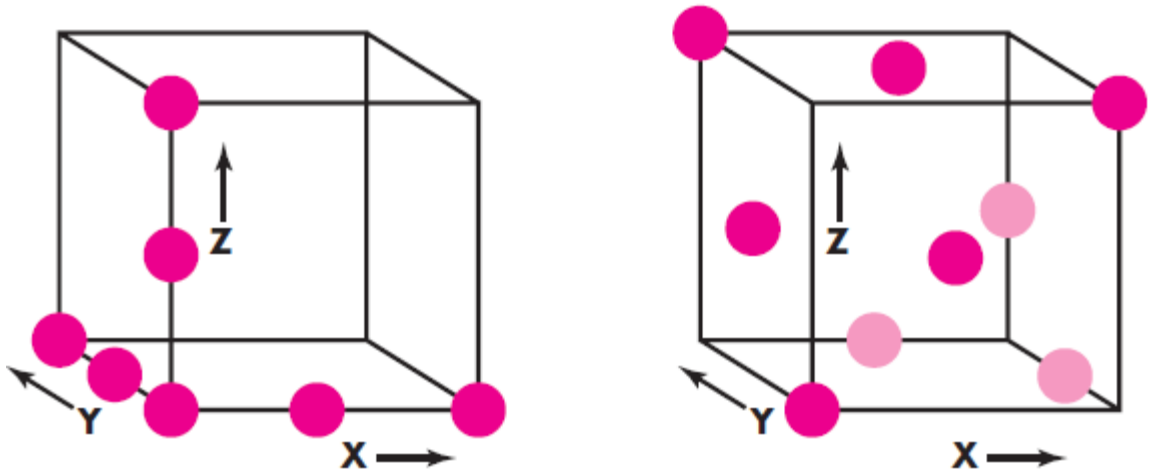

Fig 18.9: A geometric view of test cases

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

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

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

Fig 18.10: An L9 Orthogonal array

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

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

5.9.7 MODEL-BASED TESTING

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

4. Execute the test cases. Tests can be executed manually or a test script can be created and executed using a testing tool.
5. Compare actual and expected results and take corrective action as required.
MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

5.10 TESTING FOR SPECIALIZED ENVIRONMENTS, ARCHITECTURES, APPLICATIONS

Unique guidelines and approaches to testing are sometimes warranted when specialized environments, architectures, and applications are considered.

5.10.1 Testing GUIs: Graphical user interfaces (GUIs) will present you with interesting testing challenges. Because reusable components are now a common part of GUI development environments, the creation of the user interface has become less time consuming and more precise. But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases. Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite-state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI. Because of the large number of permutations associated with GUI operations, GUI testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.

5.10.2 Testing of Client-Server Architectures: The distributed nature of client-server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of client-server architectures and the software that resides within them onsiderably more difficult than stand-alone applications. In fact, recent industry studies indicate a significant increase in testing time and cost when client-server environments are developed.
In general, the testing of client-server software occurs at three different levels:
(1) Individual client applications are tested in a “disconnected” mode; the operation of the server and the underlying network are not considered. (2) The client software and associated server applications are tested in concert, but network operations are not explicitly exercised. (3) The complete client-server architecture, including network operation and performance, is tested. Although many different types of tests are conducted at each of these levels of detail, the following testing approaches are commonly encountered for client-server applications:
• **Application function tests**. The functionality of client applications is tested. In essence, the application is tested in stand-alone fashion in an attempt to uncover errors in its operation.
• **Server tests.** The coordination and data management functions of the server are tested. Server performance (overall response time and data throughput) is also considered.

• **Database tests.** The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated, and retrieved. Archiving is also tested.
• **Transaction tests.** A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues (e.g., transaction processing times and transaction volume).
• **Network communication tests.** These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests.

5.10.3 Testing Documentation and Help Facilities: The term software testing conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. It is important to note that testing must also extend to the third element of the software configuration—documentation.

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

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

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

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

In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically. Comprehensive test-case design methods for real-time systems continue to evolve. However, an overall four-step strategy can be proposed:

• **Task testing.** The first step in the testing of real-time software is to test each task independently. That is, conventional tests are designed for each task and executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior.
• **Behavioral testing.** Using system models created with automated tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built. Using a technique that is similar to *equivalence partitioning, events* (e.g., interrupts, control signals) are categorized for testing
Once each class of events has been tested, events are presented to the system in random order and with random frequency. The behavior of the software is examined to detect behavior errors.
• **Intertask testing.** Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
• **System testing.** Software and hardware are integrated, and a full range of system tests are conducted in an attempt to uncover errors at the software hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential.
Tests are then designed to assess the following system characteristics:
• Are interrupt priorities properly assigned and properly handled?
• Is processing for each interrupt handled correctly?
• Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?
• Does a high volume of interrupts arriving at critical times create problems in function or performance?
In addition, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

5.11 PATTERNS FOR SOFTWARE TESTING

Patterns can also be used to propose solutions to other software engineering situations—in this case, software testing. *Testing patterns* describe common testing problems and solutions that can assist you in dealing with them. Not only do testing patterns provide you with useful

guidance as testing activities commence, they also provide three additional benefits described by Marick:

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

Although these benefits are “*soft*,” they should not be overlooked. Testing patterns are described in much the same way as *design patterns.* The following three testing patterns provide representative examples:

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

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

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

5.12 TESTING OBJECT ORIENTED APPLICATIONS

What is it? The architecture of object-oriented (OO) 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.
Who does it? Object-oriented testing is performed by software engineers and testing specialists.
Why is it important? You have to execute the program before it gets to the customer with the specific intent of removing all errors, so that the customer will not experience the frustration associated with a poor-quality product. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.
What are the steps? OO testing is strategically analogous to the testing of conventional systems, but it is tactically different. Because the OO analysis and design models are similar in structure and content to the resultant OO program, "testing" is initiated with the review of these models. Once code has been generated, OO testing begins "in the small" with class testing. A series of tests are designed that exercise class operations and examine whether errors exist as one class collaborates with other classes. As classes are integrated to form a subsystem, thread-based, use-based, and cluster testing along with fault-based approaches are applied to fully exercise collaborating classes. Finally, use cases are used to uncover errors at the software validation level.
What is the work product? A set of test cases, designed to exercise classes, their collaborations, and behaviors is designed and documented; expected results are defined, and actual results are recorded.
How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion, and review the tests cases you do create for thoroughness.

5.12.1.BROADENING THE VIEW OF TESTING: The construction of object-oriented software begins with the creation of requirements (analysis) and design models. At each stage, the models can be "tested" in an attempt to uncover errors prior to their propagation to the next iteration.

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code levels. For example, consider a class in which a number of attributes are defined during the first iteration of analysis. An extraneous attribute is appended to the class

(due to a misunderstanding of the problem domain). Two operations are then specified to manipulate the attribute. A review is conducted and a domain expert points out the problem. By eliminating the extraneous attribute at this stage, the following problems and unnecessary effort may be avoided during analysis:

1. Special subclasses may have been generated to accommodate the *unnecessary attribute* or *exceptions* to it. Work involved in the creation of unnecessary subclasses has been avoided.
2. A misinterpretation of the class definition may lead to *incorrect* or *extraneous class relationships*.
3. The *behavior of the system* or its classes may be *improperly characterized* to accommodate the extraneous attribute.

If the problem is not uncovered during analysis and propagated further, the following problems could occur during design:
1. Improper allocation of the class to subsystem and/or tasks may occur during system design.
2. Unnecessary design work may be expended to create the procedural design for the operations that address the extraneous attribute.
3. The messaging model will be incorrect.

During latter stages of their development, object-oriented analysis (OOA) and design (OOD) models provide substantial information about the structure and behavior of the system. For this reason, these models should be subjected to rigorous review prior to the generation of code.
All object-oriented models should be tested for correctness, completeness, and consistency within the context of the model's syntax, semantics, and pragmatics.

5.12.2 TESTING OOA AND OOD MODELS

Analysis and design models cannot be tested in the conventional sense, because they cannot be executed. However, technical reviews can be used to examine their correctness and consistency.

5.12.2.1 Correctness of OOA and OOD Models: The notation and syntax used to represent analysis and design models will be tied to the specific analysis and design methods that are chosen for the project. Hence syntactic correctness is judged on proper use of the symbology; each model is reviewed to ensure that proper modeling conventions have been maintained.
During analysis and design, you can assess *semantic correctness* based on the model's conformance to the real-world problem domain. If the model accurately reflects the real world then it is semantically correct. To determine whether the model does, in fact, reflect real-world requirements, it should be presented to problem domain experts who will examine the class definitions and hierarchy for omissions and ambiguity. Class relationships are evaluated to determine whether they accurately reflect real-world object connections.

5.12.2.2 Consistency of Object-Oriented Models: The consistency of object-oriented models may be judged by "considering the relationships among entities in the model. To assess consistency, you should examine each class and its connections to other classes. The *class-responsibility-collaboration (CRC) model or an objectrelationship diagram* can be used to facilitate this activity. Each CRC card lists the class name, its responsibilities (operations), and its collaborators. The collaborations imply a series of relationships (i.e., connections) between classes of the OO system. The object relationship model provides a graphic representation of the connections between classes. All of this information can be obtained from the analysis model.

To evaluate the *class model* the following steps have been recommended:

1. Revisit the CRC model and the object-relationship model. Cross-check to ensure that all collaborations implied by the requirements model are properly reflected in the both.

2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition. For example, consider a class defined for a point-of-sale checkout system and called CreditSale. This class has a CRC index card as illustrated in Figure 19.1.

class name: credit sale	
class type: transaction event	
class characteristics: nontangible, atomic, sequential, permanent, guarded	
responsibilities:	collaborators:
read credit card	credit card
get authorization	credit authority
post purchase amount	product ticket
	sales ledger
	audit file
generate bill	bill

Fig 19.1: An example CRC index card used for review

For this collection of classes and collaborations, ask whether a responsibility (e.g., read credit card) is accomplished if delegated to the named collaborator (CreditCard). That is, does the class **CreditCard** have an operation that enables it to be read? In this case the answer is "yes." The object-relationship is traversed to ensure that all such connections are valid.

3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source. For example, if the **CreditCard** class receives

a request for purchase amount from the **CreditSale** class, there would be a problem. **CreditCard** does not know the purchase amount.

4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.

5. Determine whether widely requested responsibilities might be combined into a single responsibility. For example, read credit card and get authorization occur in every situation. They might be combined into a validate credit request responsibility that incorporates getting the credit card number and gaining authorization.

You should apply steps 1 through 5 iteratively to each class and through each evolution of the requirements model.

Once the design model is created, you should also conduct reviews of the system design and the object design.. The system design is reviewed by examining the object-behavior model developed during object-oriented analysis and mapping required system behavior against the subsystems designed to accomplish this behavior. *Concurrency* and *task allocation* are also reviewed within the context of system behavior. The *behavioral states* of the system are evaluated to determine which exist concurrently.

The object model should be tested against the object-relationship network to ensure that all design objects contain the necessary attributes and operations to implement the collaborations defined for each CRC index card.

5.12.3 OBJECT-ORIENTED TESTING STRATEGIES

5.12.3.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. *Encapsulation* drives the definition of classes and objects. This means that each class and each instance of a *class* (object) packages *attributes* (data) and the *operations* (also known as methods or services) that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class. 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 meaning of unit testing changes dramatically.

We 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 each 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 vacuum (the traditional unit-testing approach) is 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 flows 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.

5.12.3.2 Integration Testing in the OO Context: Because object-oriented software does not have a hierarchical control structure, conventional top-down and bottom-up integration strategies 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".

There are two different strategies for integration testing of OO systems. 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) of 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. Unlike conventional integration, the use of driver and stubs as replacement operations is to be avoided, when possible.

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

5.12.3.3 Validation Testing in an OO Context: At the validation or system level, the details of class connections disappear. Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable outputs from the system. To assist in the derivation of validation tests, the tester should draw upon use cases that are part of the requirements model. The use case provides a scenario that has a high likelihood of uncovered errors in user-interaction requirements.

Conventional black-box testing methods can be used to drive validation tests. In addition, you may choose to derive test cases from the objectbehavior model and from an event flow diagram created as part of OOA.

5.12.4 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 OOtest-case design has been suggested by Berard

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.

5.12.4.1 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 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. It is 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. 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.

5.12.4.2 Applicability of Conventional Test-Case Design Methods: The white-box testing methods 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.

5.12.4.3 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. 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.

It is important to note that integration testing attempts to find errors in the client object, not the server. Stated in conventional terms, the focus of integration testing is to determine whether errors exist in the calling code, not the called code. The operation call is used as a clue, a way to find test requirements that exercise the calling code.

5.12.4.4 Test Cases and the Class Hierarchy: *Inheritance* 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.
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(), 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. Test inputs may be appropriate for both base and derived classes, but the expected results may differ in the derived class.

5.12.4.5 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.

5.12.4.6 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.

For example, in a conventional system with a command-oriented interface, the user might use the list of all commands as a testing checklist. 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.

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.

5.12.5 TESTING METHODS APPLICABLE AT THE CLASS LEVEL

Testing "in the small" *focuses on a single class* and the *methods* that are encapsulated by the class. *Random testing* and *partitioning* are methods that can be used to exercise a class during OO testing.

5.12.5.1 Random Testing for OO Classes: To provide brief illustrations of these methods, consider a banking application in which an Account class has the following operations: open(), setup(), deposit(), withdraw(), balance(), summarize(), creditLimit(), and close(). Each of these operations may be applied for Account, but certain constraints are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations.

5.12.5.2 Partition Testing at the Class Level: Partition testing reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning for traditional software. Input and output are categorized and test cases are designed to exercise each category. But how are the partitioning categories derived?
State-based partitioning categorizes class operations based on their ability to change the state of the class.
Attribute-based partitioning categorizes class operations based on the attributes that they use. For the Account class, the attributes balance and creditLimit can be used to define partitions. Operations are divided into three partitions: (1) operations that use creditLimit, (2) operations that modify creditLimit, and (3) operations that do not use or modify creditLimit. Test sequences are then designed for each partition.
Category-based partitioning categorizes class operations based on the *generic function* that each performs. For example, operations in the Account class can be categorized in initialization operations (open, setup), computational operations (deposit, withdraw), queries (balance, summarize, creditLimit), and termination operations (close).

5.13 INTERCLASS TEST-CASE DESIGN

Test-case design becomes more complicated as integration of the object-oriented system begins*. It is at this stage that testing of collaborations between classes must begin*.

Like the testing of individual classes, class collaboration *testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.*

5.13.1 Multiple Class Testing: Kirani and Tsai suggest the following sequence of steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.

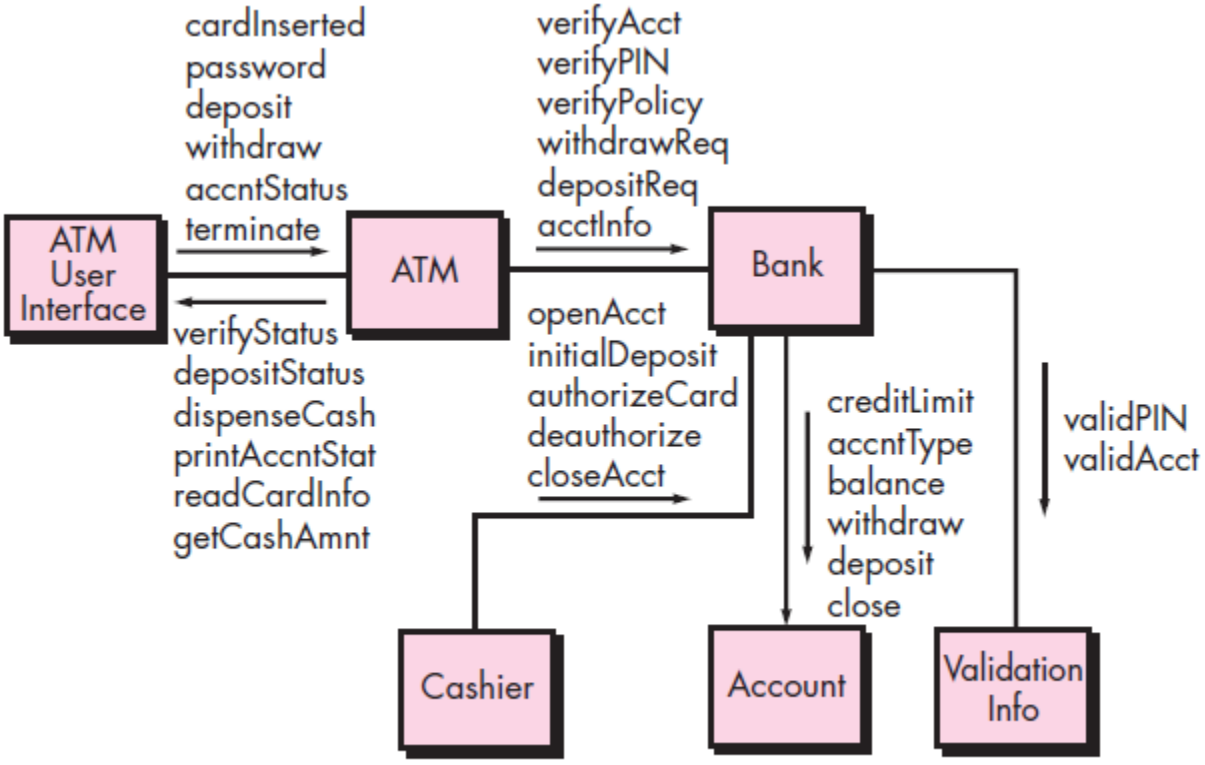

Fig 19.2: Class Collaboration diagram for banking application

2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

5.13.2 Tests Derived from Behavior Models: The use of the *state diagram* as a model that represents the dynamic behavior of a class. Figure 19.3 illustrates a state diagram for the *Account class*.

The tests to be designed should achieve coverage of every state. That is, the operation sequences should cause the Account class to make transition through all allowable states:

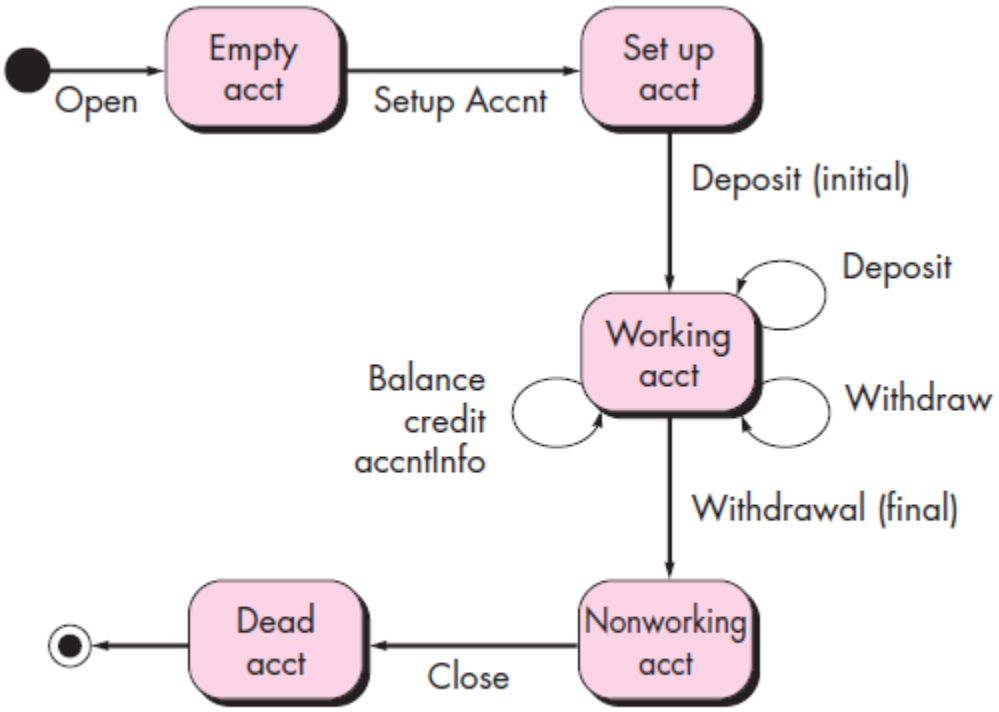

Fig 19.3: State diagram for the Account class

Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised.

The state model can be traversed in a "*breadth-first*" manner. In this context, breadth-first implies that a test case exercises a single transition and that when a new transition is to be tested only previously tested transitions are used.