- A burglar alarm system is primarily a monitoring system. It collects data from sensors but no real-time actuator control
- Control systems are similar but, in response to sensor values, the system sends control signals to actuators
- An example of a monitoring and control system is a system which monitors temperature and switches heaters on and off

UNIT IV

TESTING

Taxonomy of Software Testing

- Classified by purpose, software testing can be divided into: correctness testing, performance testing, and reliability testing and security testing.
- Classified by life-cycle phase, software testing can be classified into the following categories: requirements phase testing, design phase testing, program phase testing, evaluating test results, installation phase testing, acceptance testing and maintenance testing.
- By scope, software testing can be categorized as follows: unit testing, component testing, integration testing, and system testing.

Correctness testing

Correctness is the minimum requirement of software, the essential purpose of testing. It is used to tell the right behavior from the wrong one. The tester may or may not know the inside details of the software module under test, e.g. control flow, data flow, etc. Therefore, either a white-box point of view or black-box point of view can be taken in testing software. We must note that the black-box and white-box ideas are not limited in correctness testing only.

- Black-box testing
- White-box testing

Performance testing

Not all software systems have specifications on performance explicitly. But every system will have implicit performance requirements. The software should not take infinite time or infinite resource to execute. "Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade.

Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes: resource usage, throughput, stimulus-response time and queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources. Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage. The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc.

Reliability testing

Software reliability refers to the probability of failure-free operation of a system. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult. Testing is an effective sampling method to measure software reliability. Guided by the operational profile, software testing (usually black-box testing) can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability. Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software. Risk of using software can also be assessed based on reliability information.

Security testing

Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe.
Many critical software applications and services have integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to find vulnerabilities.

Types of S/W Test

Acceptance testing

Testing to verify a product meets customer specified requirements. A customer usually does this type of testing on a product that is developed externally.

Compatibility testing

This is used to ensure compatibility of an application or Web site with different browsers, OSs, and hardware platforms. Compatibility testing can be performed manually or can be driven by an automated functional or regression test suite.

Conformance testing

This is used to verify implementation conformance to industry standards. Producing tests for the behavior of an implementation to be sure it provides the portability, interoperability, and/or compatibility a standard defines.

Integration testing

Modules are typically code modules, individual applications, client and server applications on a network, etc. Integration Testing follows unit testing and precedes system testing.

Load testing

Load testing is a generic term covering Performance Testing and Stress Testing.

Performance testing

Performance testing can be applied to understand your application or WWW site's scalability, or to benchmark the performance in an environment of third party products such as servers and middleware for potential purchase. This sort of testing is particularly useful to identify performance bottlenecks in high use applications. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions.

Regression testing

Similar in scope to a functional test, a regression test allows a consistent, repeatable validation of each new release of a product or Web site. Such testing ensures reported product defects have been corrected for each new release and that no new quality problems were introduced in the maintenance process. Though regression testing can be performed manually an automated test suite is often used to reduce the time and resources needed to perform the required testing.

System testing

Entire system is tested as per the requirements. Black-box type testing that is based on overall requirements specifications, covers all combined parts of a system.

End-to-end testing

Similar to system testing, involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Sanity testing

Testing is to determine if a new software version is performing well enough to accept it for a major testing effort. If application is crashing for initial use then system is not stable enough for further testing and build or application is assigned to fix.

Alpha testing

In house virtual user environment can be created for this type of testing. Testing is done at the end of development. Still minor design changes may be made as a result of such testing.

Beta testing

Testing is typically done by end-users or others. This is the final testing before releasing the application to commercial purpose.

Software Testing Techniques

Software Testing:

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Testing Objectives:

- Testing is the process of executing a program with the intent of finding errors.
- A good test case is one with a high probability of finding an as-yet undiscovered error.
- A successful test is one that discovers an as-yet-undiscovered error.

Testing Principles:

- All tests should be traceable to customer requirements.
- Tests should be planned before testing begins.
- 80% of all errors are in 20% of the code.
- Testing should begin in the small and progress to the large.
- Exhaustive testing is not possible.

Testing should be conducted by an independent third party if possible.

Software Defect Causes:

- Specification may be wrong.
- Specification may be a physical impossibility.
- Faulty program design.
- Program may be incorrect.

Types of Errors:

- Algorithmic error.
- Computation & precision error.
- Documentation error.
- Capacity error or boundary error.
- Timing and coordination error.
- Throughput or performance error.
- Recovery error.
- Hardware & system software error.
- Standards & procedure errors.

Software Testability Checklist – 1:

- Operability
  - if it works better it can be tested more efficiently
- Observability
  - what you see is what you test
- Controllability
  - if software can be controlled better the it is more that testing can be automated and optimized

Software Testability Checklist – 2:

- Decomposability
  - controlling the scope of testing allows problems to be isolated quickly and retested intelligently
- Stability
  - the fewer the changes, the fewer the disruptions to testing
- Understandability
  - the more information that is known, the smarter the testing can be done

Good Test Attributes:

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be best of breed.
- A good test should not be too simple or too complex.

Test Strategies:

- Black-box or behavioral testing

  - knowing the specified function a product is to perform and demonstrating correct operation based solely on its specification without regard for its internal logic
- White-box or glass-box testing
  - knowing the internal workings of a product, tests are performed to check the workings of all possible logic paths

White-Box Testing:

Basis Path Testing:

- White-box technique usually based on the program flow graph
- The cyclo matic complexity of the program computed from its flow graph using the formula V(G) = E – N + 2 or by counting the conditional statements in the PDL representation and adding 1
- Determine the basis set of linearly independent paths (the cardinality of this set is the program cyclomatic complexity)
- Prepare test cases that will force the execution of each path in the basis set.

Cyclomatic Complexity:

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.

Control Structure Testing – 1:

- White-box techniques focusing on control structures present in the software
- **Condition testing (e.g. branch testing)**
  - focuses on testing each decision statement in a software module
  - it is important to ensure coverage of all logical combinations of data that may be processed by the module (a truth table may be helpful)

Control Structure Testing – 2:

- Data flow testing
  - selects test paths based according to the locations of variable definitions and uses in the program (e.g. definition use chains)
- Loop testing
  - focuses on the validity of the program loop constructs (i.e. while, for, go to)
  - involves checking to ensure loops start and stop when they are supposed to (unstructured loops should be redesigned whenever possible)

Loop Testing: Simple Loops:

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m < n
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Loop Testing: Nested Loops:

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
  then treat each as a simple loop
  else* treat as nested loops
end if*
for example, the final loop counter value of loop 1 is
used to initialize loop 2.

Black-Box Testing:

Graph-Based Testing – 1:

- Black-box methods based on the nature of the relationships (links) among the program objects (nodes), test cases are designed to traverse the entire graph
- Transaction flow testing
  - nodes represent steps in some transaction and links represent logical connections between steps that need to be validated
- Finite state modeling
  - nodes represent user observable states of the software and links represent state transitions

Graph-Based Testing – 2:

- Data flow modeling
  - nodes are data objects and links are transformations of one data object to another data object
- Timing modeling
  - nodes are program objects and links are sequential connections between these objects
  - link weights are required execution times

Equivalence Partitioning:

- Black-box technique that divides the input domain into classes of data from which test cases can be derived
- An ideal test case uncovers a class of errors that might require many arbitrary test cases to be executed before a general error is observed

Equivalence Class Guidelines:

- If input condition specifies a range, one valid and two invalid equivalence classes are defined
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
- If an input condition specifies a member of a set, one valid and one invalid equivale nce class is defined
- If an input condition is Boolean, one valid and one invalid equivalence class is defined
- Boundary Value Analysis - 1
- Black-box technique
  - focuses on the boundaries of the input domain rather than its center
- Guidelines:

- If input condition specifies a range bounded by values a and b, test cases should include a and b, values just above and just below a and b
- If an input condition specifies and number of values, test cases should be exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values

Boundary Value Analysis – 2

1. Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maximum output reports
2. If internal program data structures have boundaries (e.g. size limitations), be certain to test the boundaries

Comparison Testing:

- Black-box testing for safety critical systems in which independently developed implementations of redundant systems are tested for conformance to specifications
- Often equivalence class partitioning is used to develop a common set of test cases for each implementation

Orthogonal Array Testing – 1:

- Black-box technique that enables the design of a reasonably small set of test cases that provide maximum test coverage
- Focus is on categories of faulty logic likely to be present in the software component (without examining the code)

Orthogonal Array Testing – 2:

- Priorities for assessing tests using an orthogonal array
  - Detect and isolate all single mode faults
  - Detect all double mode faults
  - Multimode faults

Software Testing Strategies:

Strategic Approach to Testing – 1:

- Testing begins at the component level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- The developer of the software conducts testing and may be assisted by independent test groups for large projects.
- The role of the independent tester is to remove the conflict of interest inherent when the builder is testing his or her own product.

Strategic Approach to Testing – 2:

- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.
- Need to consider verification issues
  - are we building the product right?
- Need to Consider validation issuesare we building the right product?

Verification vs validation:

- Verification:
  "Are we building the product right" The software should conform to its specification
  Validation:

"Are we building the right product" The software should do what the user really requires

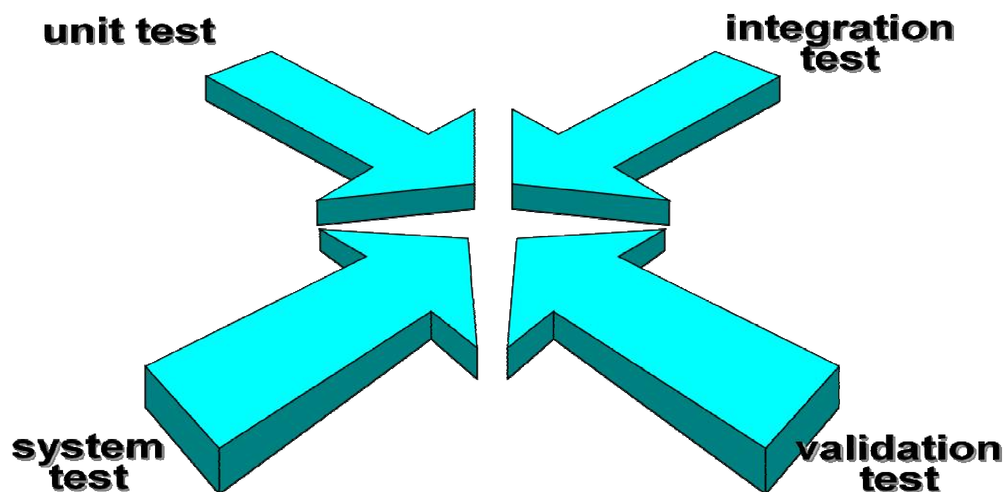
The V & V process:

- As a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation.
- Strategic Testing Issues - 1 Specify product requirements in a quantifiable manner before testing starts.
- Specify testing objectives explicitly.
- Identify the user classes of the software and develop a profile for each.
- Develop a test plan that emphasizes rapid cycle testing.

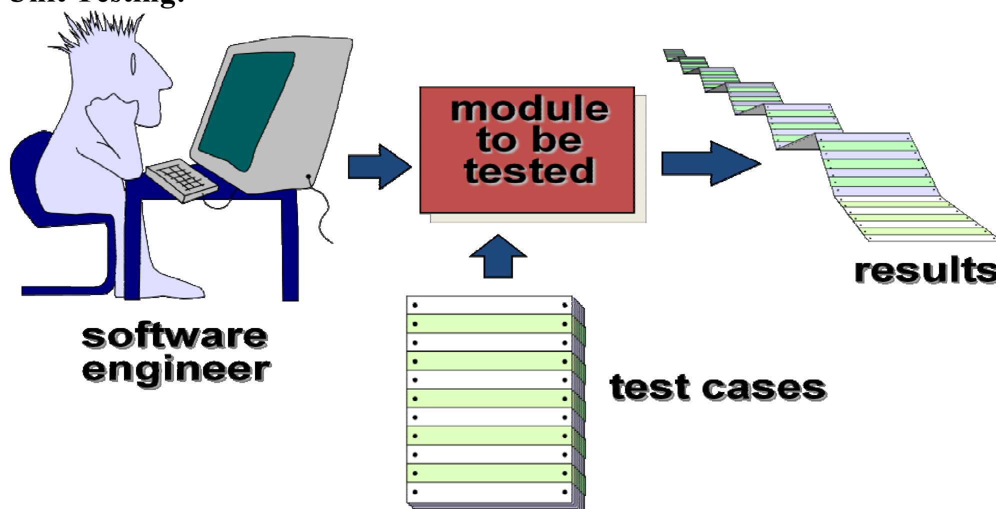
Strategic Testing Issues – 2:

- Build robust software that is designed to test itself (e.g. use anti-bugging).
- Use effective formal reviews as a filter prior to testing.
- Conduct formal technical reviews to assess the test strategy and test cases.

Testing Strategy:



Unit Testing:



- Program reviews.
- Formal verification.
- Testing the program itself.
    - black box and white box testing.

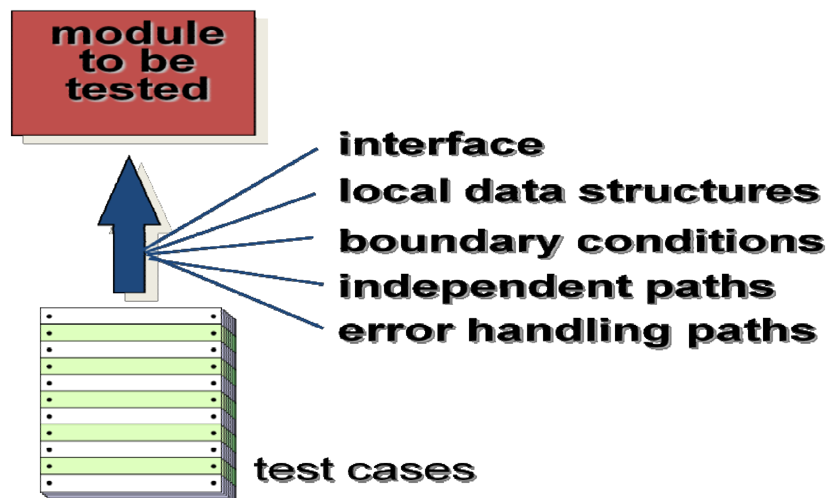
Black Box or White Box?:

- Maximum # of logic paths - determine if white box testing is possible.
- Nature of input data.
- Amount of computation involved.
- Complexity of algorithms.

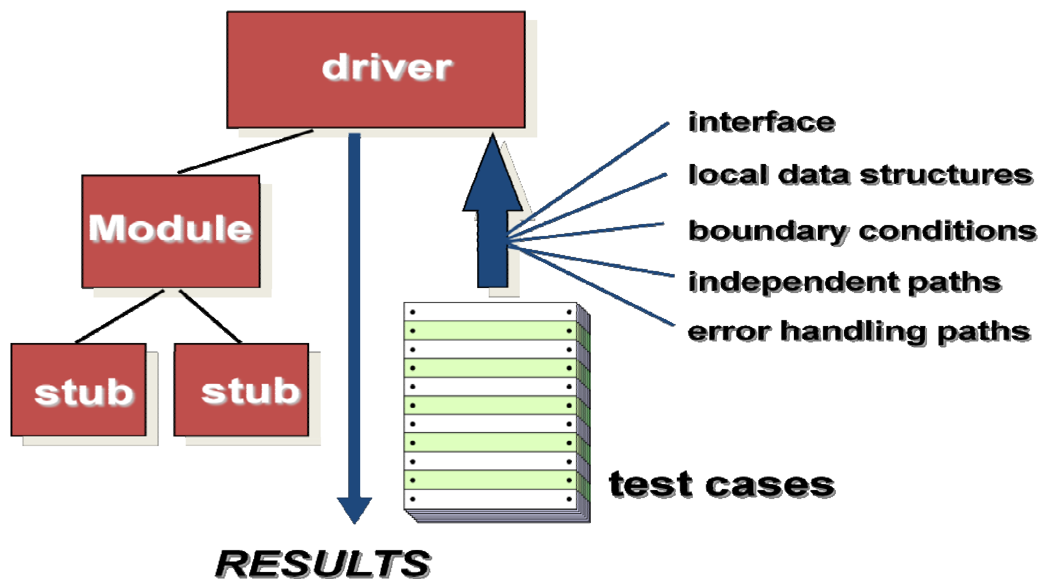
Unit Testing Details:

- Interfaces tested for proper information flow.
- Local data are examined to ensure that integrity is maintained.
- Boundary conditions are tested.
- Basis path testing should be used.
- All error handling paths should be tested.
- Drivers and/or stubs need to be developed to test incomplete software.

Unit Testing:



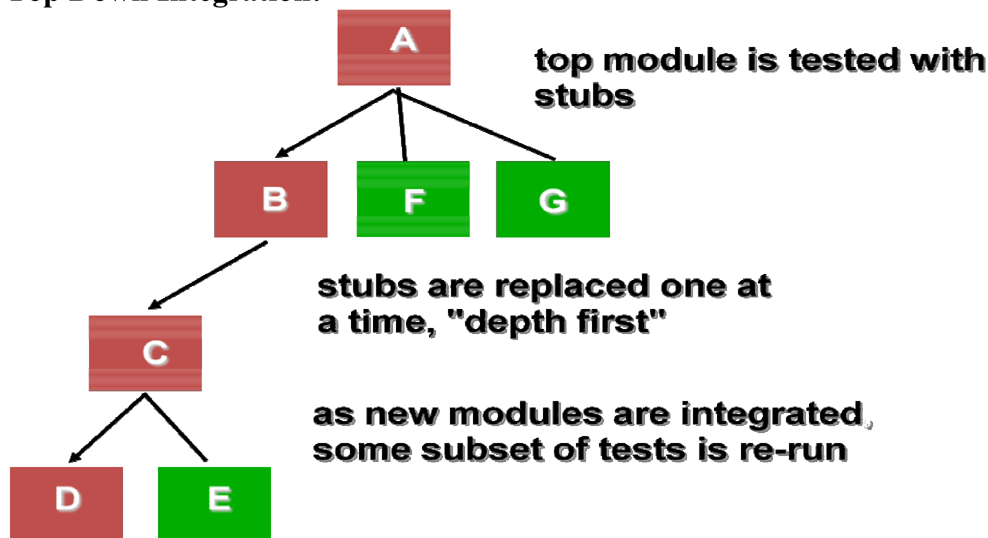
Unit Test Environment:



Integration Testing:

- Bottom - up testing (test harness).
- Top - down testing (stubs).
- Regression Testing.
- Smoke Testing

Top Down Integration:

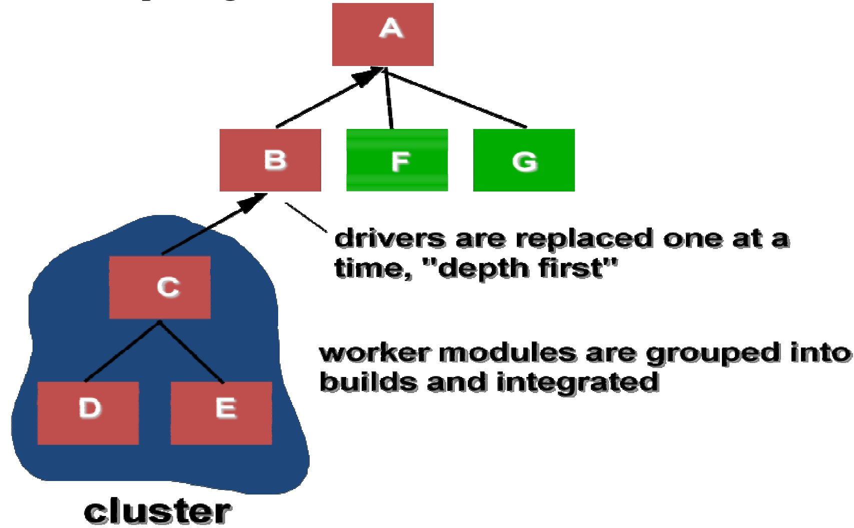


Top-Down Integration Testing:

- Main program used as a test driver and stubs are substitutes for components directly subordinate to it.
- Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
- Tests are conducted as each component is integrated.
- On completion of each set of tests and other stub is replaced with a real component.

- Regression testing may be used to ensure that new errors not introduced.

Bottom-Up Integration:



Bottom-Up Integration Testing:

- Low level components are combined in clusters that perform a specific software function.
- A driver (control program) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.

Regression Testing:

- The selective retesting of a software system that has been modified to ensure that any bugs have been fixed and that no other previously working functions have failed as a result of the reparations and that newly added features have not created problems with previous versions of the software. Also referred to as verification testing, regression testing is initiated after a programmer has attempted to fix a recognized problem or has added source code to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.

Regression Testing:

- Regression test suit contains 3 different classes of test cases
  - Representative sample of existing test cases is used to exercise all software functions.
  - Additional test cases focusing software functions likely to be affected by the change.
  - Tests cases that focus on the changed software components.

Smoke Testing:

- Software components already translated into code are integrated into a build.
- A series of tests designed to expose errors that will keep the build from performing its functions are created.
- The build is integrated with the other builds and the entire product is smoke tested daily using either top-down or bottom integration.

Validation Testing:

- Ensure that each function or performance characteristic conforms to its specification.
- Deviations (deficiencies) must be negotiated with the customer to establish a means for resolving the errors.
- Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Acceptance Testing:

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test
  - version of the complete software is tested by customer under the supervision of the developer at the developer‘s site
- Beta test
  - version of the complete software is tested by customer at his or her own site without the developer being present

System Testing:

- Recovery testing
  - checks system‘s ability to recover from failures
- Security testing
  - verifies that system protection mechanism prevents improper penetration or data alteration
- Stress testing
  - program is checked to see how well it deals with abnormal resource demands
- Performance testing
  - tests the run-time performance of software

Performance Testing:

- Stress test.
- Volume test.
- Configuration test (hardware & software).
- Compatibility.
- Regression tests.
- Security tests.
- Timing tests.
- Environmental tests.
- Quality tests.
- Recovery tests.
- Maintenance tests.
- Documentation tests.
- Human factors tests.

Testing Life Cycle:

- Establish test objectives.
- Design criteria (review criteria).
  - Correct.
  - Feasible.
  - Coverage.
  - Demonstrate functionality.

- Writing test cases.
- Testing test cases.
- Execute test cases.
- Evaluate test results.

Testing Tools:

- Simulators.
- Monitors.
- Analyzers.
- Test data generators.

Document Each Test Case:

- Requirement tested.
- Facet / feature / path tested.
- Person & date.
- Tools & code needed.
- Test data & instructions.
- Expected results.
- Actual test results & analysis
- Correction, schedule, and signoff.

Debugging:

- Debugging (removal of a defect) occurs as a consequence of successful testing.
- Some people better at debugging than others.
- Is the cause of the bug reproduced in another part of the program?
- What ‒next bug‖ might be introduced by the fix that is being proposed?
- What could have been done to prevent this bug in the first place?

Software Implementation techniques

- Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.
- Software Implementation Techniques include process and thread scheduling, synchronization and concurrency primitives, file management, memory management, performance, networking facilities, and user interfaces. Software Implementation Techniques is designed to facilitate determining what is required to implement a specific operating system function.

Procedural programming

Procedural programming can sometimes be used as a synonym for imperative programming (specifying the steps the program must take to reach the desired state), but can also refer (as in this article) to a programming paradigm, derived from structured programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, methods, or functions (not to be confused with mathematical functions, but similar to those used in functional programming) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself. Some good examples of procedural programs are the Linux Kernel, GIT, Apache Server, and Quake III Arena.

Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. Many modern programming languages now support OOP.

An object-oriented program may thus be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable o f receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to 'carry their own operators around with them' (or at least "inherit" them from a similar object or class). In the conventional model, the data and operations on the data don't have a tight, formal association.

functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

In practice, the difference between a mathematical function and the notion of a "function" used in imperative programming is that imperative functions can have side effects, changing the value of already calculated computations. Because of this they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program. Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result $f(x)$ both times. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.JavaScript, one of the most widely employed languages today, incorporates functional programming capabilities.

Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, which can be traced at least as far back as John McCarthy's [1958] advice-taker proposal, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.

Oracle's Application Implementation Method

AIM provides with an integrated set of templates, procedures, PowerPoint presentations, spreadsheets, and project plans for implementing the applications. AIM was such a success, Oracle created a subset of the templates, called it AIM Advantage, and made it available as a product to customers and other consulting firms. Since its initial release, AIM has been revised and improved several times with new templates and methods.

AIM Is a Six-Phase Method

Because the Oracle ERP Applications are software modules buy from a vendor, different implementation methods are used than the techniques used for custom developed systems. AIM has six major phases:

- **Definition phase**: During this phase, you plan the project, determine business objectives, and verify the feasibility of the project for given time, resource, and budget limits.
- **Operations Analysis phase**: Includes documents business requirements, gaps in the software (which can lead to customizations), and system architecture requirements. Results of the analysis should provide a proposal for future business processes, a technical architecture model, an application architecture model, workarounds for application gaps, performance testing models, and a transition strategy to migrate to the new systems. Another task that can begin in this phase is mapping of legacy data to Oracle Application APIs or open interfaces—data conversion.
- **Solution Design phase**—Used to create designs for solutions that meet future business requirements and processes. The design of your future organization comes alive during this phase as customizations and module configurations are finalized.
- **Build phase**—During this phase of AIM, coding and testing of customizations, enhancements, interfaces, and data conversions happens. In addition, one or more conference room pilots test the integrated enterprise system. The results of the build phase should be a working, tested business system solution.
- **Transition phase**—During this phase, the project team delivers the finished solution to the enterprise. End-user training and support, management of change, and data conversions are major activities of this phase.
- **Production phase**—Starts when the system goes live. Technical people work to stabilize and maintain the system under full transaction loads. Users and the implementation team begin a series of refinements to minimize unfavorable impacts and realize the business objectives identified in the definition phase.

Rapid Implementations

In the late 1990s as Y2K approached, customers demanded and consulting firms discovered faster ways to implement packaged software applicat ions. The rapid implementation became possible for certain types of customers. The events that converged in the late 1990s to provide faster implementations include the following:

- Many smaller companies couldn‘t afford the big ERP project. If the software vendors and consulting firms were going to sell to the ‒middle market‖ companies, they had to develop more efficient methods.
- Many dotcoms needed a financial infrastructure; ERP applications filled the need, and rapid implementation methods provided the way.
- The functionality of the software improved a lot, many gaps were eliminated, and more companies could implement with fewer customizations.
- After the big, complex companies implemented their ERP systems, the typical implementation became less difficult.
- The number of skilled consultants and project managers increased significantly.
- Other software vendors started packaging preprogrammed integration points to the Oracle ERP modules.

Rapid implementations focus on delivering a predefined set of functionality. A key set of business processes is installed in a standard way to accelerate the implementation schedule. These projects benefit from the use of preconfigured modules and predefined business processes. You get to reuse the analysis and integration testing from other implementations, and you agree to ignore all gaps by modifying your business to fit the software. Typically, the enterprise will be allowed some control over key decisions such as the structure of the chart of accounts. Fixed budgets are set for training, production support, and data conversions (a limited amount of data).

Phased Implementations

Phased implementations seek to break up the work of an ERP implementation project. This technique can make the system more manageable and reduce risks, and costs in some cases, to the enterprise. In the mid-1990s, 4 or 5 was about the maximum number of application modules that could be launched into production at one time. If you bought 12 or 13 applications, there would be a financial phase that would be followed by phases for the distribution and manufacturing applications. As implementation techniques improved and Y2K pressures grew in the late 1990s, more and more companies started launching most of their applications at the same time. This method became known as the big-bang approach. Now, each company selects a phased or big-bang approach based on its individual requirements.

Another approach to phasing can be employed by companies with business units at multiple sites. With this technique, one business unit is used as a template, and all applications are completely implemented in an initial phase lasting 10–14 months. Then, other sites implement the applications in cookie-cutter fashion. The cookie-cutter phases are focused on end-user training and the differences that a site has from the prototype site. The cookie-cutter phase can be as short as 9–12 weeks, and these phases can be conducted at several sites simultaneously. For your reference, we participated in an efficient project where 13 app lications were implemented big bang–style in July at the Chicago site after about 8 months work. A site in Malaysia went live in October. The Ireland site started up in November. After a holiday break, the Atlanta business unit went live in February, and the final site in China started using the applications in April. Implementing thirteen application modules at five sites in four countries in sixteen months was pretty impressive.

Case Studies Illustrating Implementation Techniques

Some practical examples from the real world might help to illustrate some of the principles and techniques of various software implementation methods. These case studies are composites from about 60 implementation projects we have observed during the past 9 years.

Big companies often have a horrible time resolving issues and deciding on configuration parameters because there is so much money involved and each of many sites might want to control decisions about what it considers its critical success factors. For example, we once saw a large company argue for over two months about the chart of accounts structure, while eight consultants from two consulting firms tried to referee among the feuding operating units. Another large company labored for more than six months to unify a mast er customer list for a centralized receivables and decentralized order entry system.

Transition activities at large companies need special attention. Training end users can be a logistical challenge and can require considerable planning. For example, if you have 800 users to train and each user needs an average of three classes of two hours each and you have one month, how many classrooms and instructors do you need? Another example is that loading data

from a legacy system can be a problem. If you have one million customers to load into Oracle receivables at the rate of 5,000/hour and the database administrator allows you to load 20 hours per day, you have a 10-day task.

Because they spend huge amounts of money on their ERP systems, many big companies try to optimize the systems and capture specific returns on the investment. However, sometimes companies can be incredibly insensitive and uncoordinated as they try to make money from their ERP software. For example, one business announced at the beginning of a project that the accounts payable department would be cut from 50–17 employees as soon as the system went live. Another company decided to centralize about 30 accounting sites into one shared service center and advised about 60 accountants that they would lose their jobs in about a year. Several of the 60 employees were offered positions on the ERP implementation team.

Small companies have other problems when creating an implementation team. Occasionally, the small company tries to put clerical employees on the team and they have problems with issue resolution or some of the ERP concepts. In another case, one small company didn‘t create the position of project manager. Each department worked on its own modules and ignored the integration points, testing, and requirements of other users. When Y2K deadlines forced the system startup, results were disastrous with a cost impact that doubled the cost of the entire project.

Project team members at small companies sometimes have a hard time relating to the cost of the implementation. We once worked with a company where the project manager (who was also the database administrator) advised me within the first hour of our meeting that he thought consulting charges of $3/minute were outrageous, and he couldn‘t rationalize how we could possibly make such a contribution. We agreed a consultant could not contribute $3 in value each and every minute to his project. However, when I told him we would be able to save him $10,000/week and make the difference between success and failure, he realized we should get to work.

Because the small company might be relatively simple to implement and the technical staff might be inexperienced with the database and software, it is possible that the technical staff will be on the critical path of the project. If the database administrator can‘t learn how to handle the production database by the time the users are ready to go live, you might need to hire some temporary help to enable the users to keep to the schedule. In addition, we often see small companies with just a single database administrator who might be working 60 or more hours per week. They feel they can afford to have more DBAs as employees, but they don‘t know how to establish the right ratio of support staff to user requirements. These companies can burn out a DBA quickly and then have to deal with the problem of replacing an important skill.