SYLLABUS OF SEMESTER –I, M.C.A. (Master In Computer Application)

Course Code: MCT543 Course: Concept in Software Engineering

Section -III (Weightage – 30% , Minimum Theory Teaching Hours -8)

Software Quality Management: Software quality assurance, Software testing techniques, S/W testing fundamentals, White box testing, Black box testing, Validation testing, System testing, Debugging,

software maintenance: maintainability, Maintenance tasks, Reverse engineering and reengineering, Importance of Release Engineering.

Software Quality Assurance

- **Software quality assurance** (often called *quality management*) is an umbrella activity that is applied throughout the software process.
- Software quality assurance (SQA) encompasses:

(1) an SQA process,

(2) specific quality assurance and quality control tasks (including technical reviews and a multitiered testing strategy),

(3) effective software engineering practice (methods and tools),

(4) control of all software work products and the changes made to them,

(5) a procedure to ensure compliance with software development standards (when applicable), and

(6) measurement and reporting mechanisms.

ELEMENTS OF SOFTWARE QUALITY ASSURANCE

- Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner:

1. Standards
2. Reviews and audits
3. Testing
4. Error/defect collection and analysis
5. Change management
6. Education
7. Vendor management
8. Security management
9. Safety
10. Risk management

Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. **The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.**

Reviews and audits. **Technical reviews are a quality control activity performed by software engineers for software engineers.** Their intent is to uncover errors. **Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work.** For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

Testing. Software testing is a quality control function that has one primary goal—to find errors. **The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.**

Error/defect collection and analysis. The only way to improve is to measure how you're doing. **SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.**

Change management. Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. **SQA ensures that adequate change management practices have been instituted.**

Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. **The SQA organization takes the lead in software process improvement and is a key promotor and sponsor of educational programs.**

Vendor management. Three categories of software are acquired from external software vendors—**shrink-wrapped packages** (e.g., Microsoft Office), **a tailored shell** *that provides a basic skeletal structure that is custom tailored to the needs of a purchaser,* and **contracted software** *that is custom designed and constructed from specifications provided by the customer organization.* **The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor.**

Security management. With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally. **SQA ensures that appropriate process and technology are used to achieve software security.**

Safety. Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. **SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.**

Risk management. Although the analysis and mitigation of risk is the concern of software engineers, **the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.**

SQA Tasks

The charter of the SQA group is to assist the software team in achieving a high-quality end product. **The Software Engineering Institute recommends a set of SQA actions that address quality assurance planning, oversight, record keeping, analysis, and reporting.** *These actions are performed (or facilitated) by an independent SQA group that:*

Prepares an SQA plan for a project.

The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance actions performed by the software engineering team and the SQA group are governed by the plan. **The plan identifies evaluations to be performed, audits and reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software team.**

Participates in the development of the project's software process description.

The software team selects a process for the work to be performed. **The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.**

Reviews software engineering activities to verify compliance with the defined software process.

The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process.

The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure.

Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.

Records any noncompliance and reports to senior management.

Noncompliance items are tracked until they are resolved.

Goals, Attributes, and Metrics

The SQA actions described in the preceding section are performed to achieve a set of practical goals:

Requirements quality. The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. **SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.**

Design quality. Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. **SQA looks for attributes of the design that are indicators of quality.**

Code quality. Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability. **SQA should isolate those attributes that allow a reasonable analysis of the quality of code.**

Quality control effectiveness. A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. **SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.**

Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors (Chapter 8)
	Understandability	Percent internal comments
		Variable naming conventions
	Reusability	Percent reused components
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics (Chapter 14)
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

METRICS FOR SOFTWARE QUALITY

- The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors.
- A project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are combined to provide project-level results.
- Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.
- Metrics such as work product errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the *defect removal efficiency* (DRE) for each process framework activity.

Measuring Quality

Although there are many measures of software quality, *correctness, maintainability, integrity,* and *usability* provide useful indicators for the project team.

Correctness. A program must operate correctly or it provides little value to its users. **Correctness is the degree to which the software performs its required function.** The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.

When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

Maintainability. Software maintenance and support accounts for more effort than any other software engineering activity. **Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements.** There is no way to measure maintainability directly; therefore, you must use indirect measures. A simple time-oriented metric is *mean-time-to-change* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users. **On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.**

Measuring Quality . . .

Integrity. Software integrity has become increasingly important in the age of cyber terrorists and hackers. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: *programs, data,* and *documentation*.

To measure integrity, two additional attributes must be defined: *threat* and *security*. ***Threat*** is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time.

Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as:

$$\text{Integrity} = \sum[1 - (\text{threat} \times (1 - \text{security}))]$$

Usability. If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable. **Usability is an attempt to quantify ease of use and can be measured in terms of the characteristics.**

Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is *defect removal efficiency* (DRE). In essence, *DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process frame-work activities.*

When considered for a project as a whole, DRE is defined in the following manner:

$$\text{DRE} = \frac{E}{E + D}$$

where *E* is the number of errors found before delivery of the software to the end user and *D* is the number of defects found after delivery.

The ideal value for *DRE* is 1. That is, no defects are found in the software. Realistically, *D* will be greater than 0, but the value of DRE can still approach 1 as E increases for a given value of *D*.

In fact, as *E* increases, it is likely that the final value of *D* will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, *DRE* encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering action.

For example, requirements analysis produces a requirements model that can be reviewed to find and correct errors. Those errors that are not found during the review of the requirements model are passed on to design (where they may or may not be found). When used in this context, we redefine *DRE* as

$$\mathrm{DRE}_i = \frac{E_i}{E_i + E_{i+1}}$$

where E_i is the number of errors found during software engineering action *i* and E_{i+1} is the number of errors found during software engineering action *i + 1* that are traceable to errors that were not discovered in software engineering action *i*.

A quality objective for a software team (or an individual software engineer) is to achieve DRE_i that approaches 1. That is, errors should be filtered out before they are passed on to the next activity or action.

Halstead's Software Science

- Halstead's theory of "software science" proposed the first analytical "laws" for computer software.
- Halstead assigned quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete.
- The measures are:

  n_1 = number of distinct operators that appear in a program

  n_2 = number of distinct operands that appear in a program

  N_1 = total number of operator occurrences

  N_2 = total number of operand occurrences

- Halstead uses these primitive measures to develop expressions for the overall **program length**, **potential minimum volume for an algorithm**, **the actual volume (number of bits required to specify a program)**, **the program level (a measure of software complexity)**, **the language level (a constant for a given language)**, and **other features such as *development effort, development time, and even the projected number of faults in the software***.

Halstead shows that **length *N*** can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and **program volume** may be defined

$$V = N \log_2 (n_1 + n_2)$$

- It should be noted that *V* will vary with programming language and represents the volume of information (in bits) required to specify a program.
- Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio *L* as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, *L* must always be less than 1.
- In terms of primitive measures, **the volume ratio** may be expressed as

$$L = \frac{2}{n_1} \text{ X } \frac{N_2}{n_2}$$

- Halstead's work is agreeable to experimental verification and a large body of research has been conducted to investigate software science.

SOFTWARE TESTING

- Software is tested to uncover errors that were made inadvertently as it was designed and constructed.
- Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which you can place specific test case design techniques and testing methods—should be defined for the software process.
- A strategy for software testing is developed by the project manager, software engineers, and testing specialists.
- Testing begins “in the small” and progresses “to the large.” This means 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.

A STRATEGIC APPROACH TO SOFTWARE TESTING

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

- A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:

  - To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
  - Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
  - Different testing techniques are appropriate for different software engineering approaches and at different points in time.
  - Testing is conducted by the developer of the software and (for large projects) an independent test group.
  - Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

- A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements.
- A strategy should provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

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:

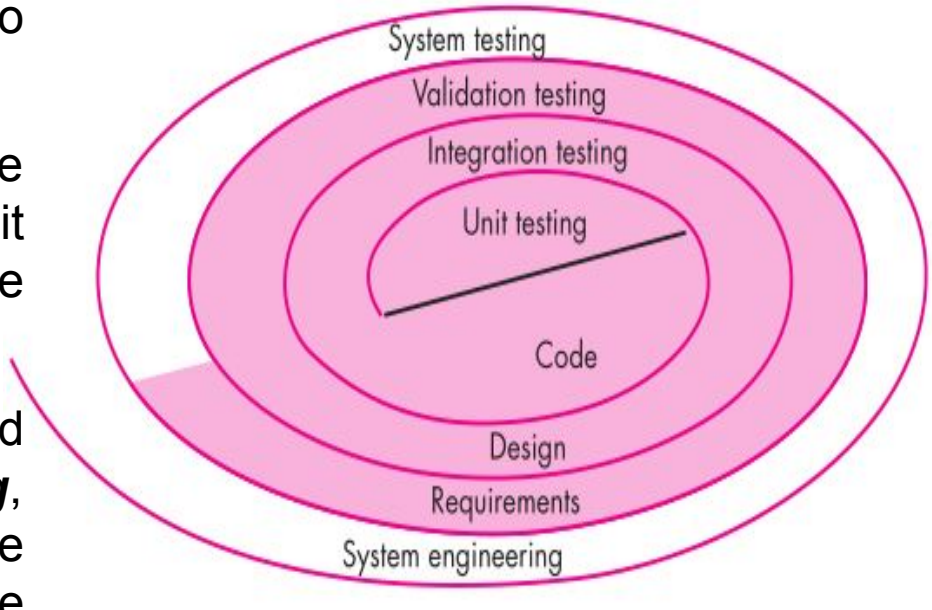
  Verification: “Are we building the product right?”

  Validation: “Are we building the right product?”

Software Testing Strategy

- The software process may be viewed as the spiral.
- Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.
- Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counterclockwise) along streamlines that decrease the level of abstraction on each turn.

- A strategy for software testing may also be viewed in the context of the spiral.
- ***Unit testing*** begins at the vortex of the spiral and concentrates on each unit (e.g., component, class) 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.

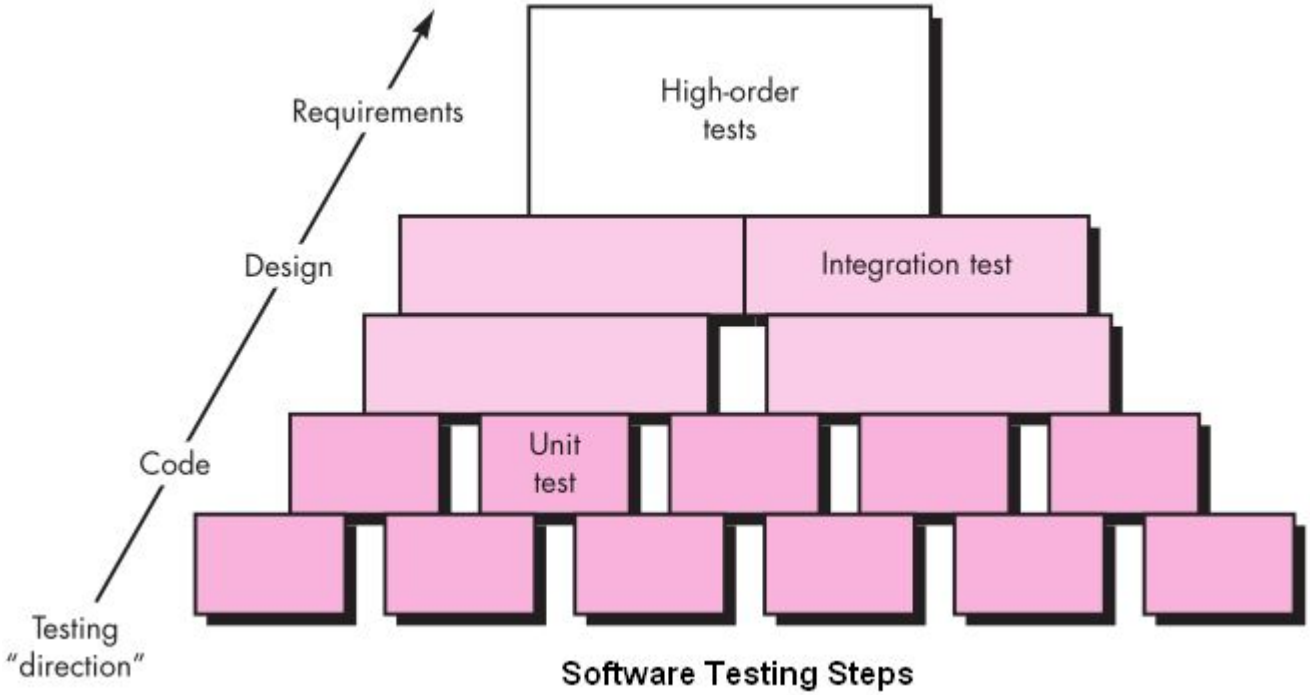


Testing Strategy

- 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.

Software Testing Strategy …

- 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.
- Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing***. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.



Software Testing Steps

- 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.
- 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.

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.

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. Kaner, Falk, and Nguyen suggest the following attributes of a "good" test:

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

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

A good test should be "best of breed". 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.

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.

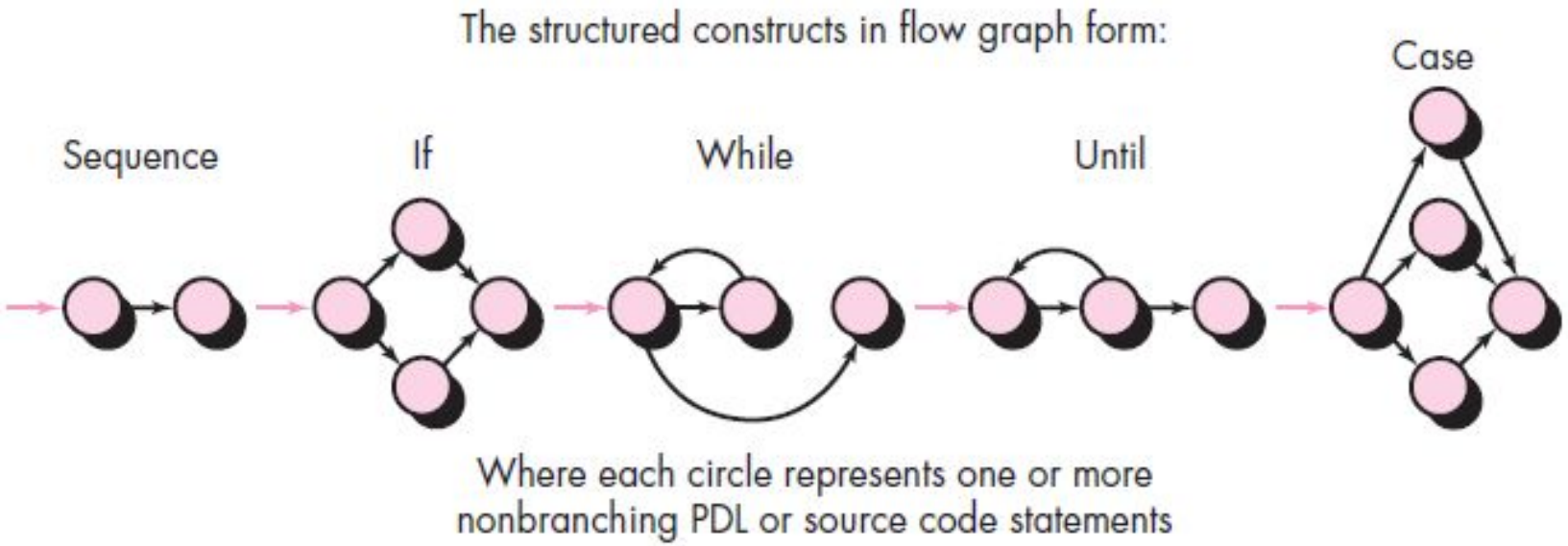
- ***White-box testing*** of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

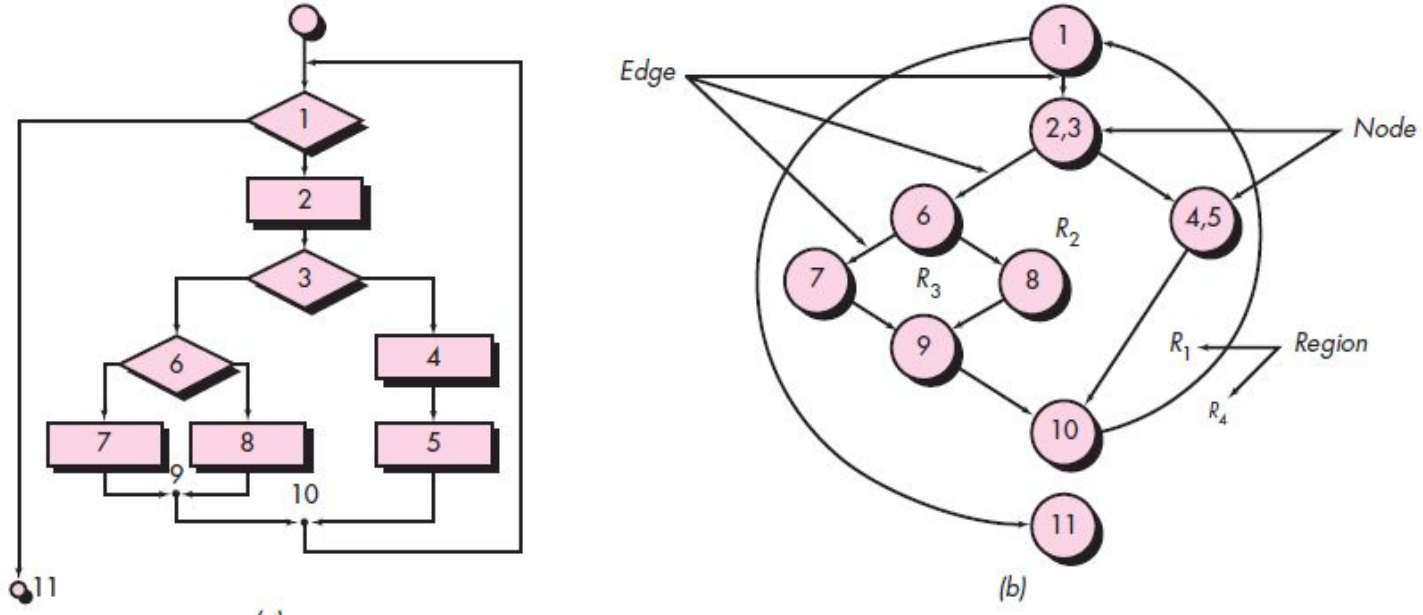
- Using white-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

- ***Basis path testing*** 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.

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.

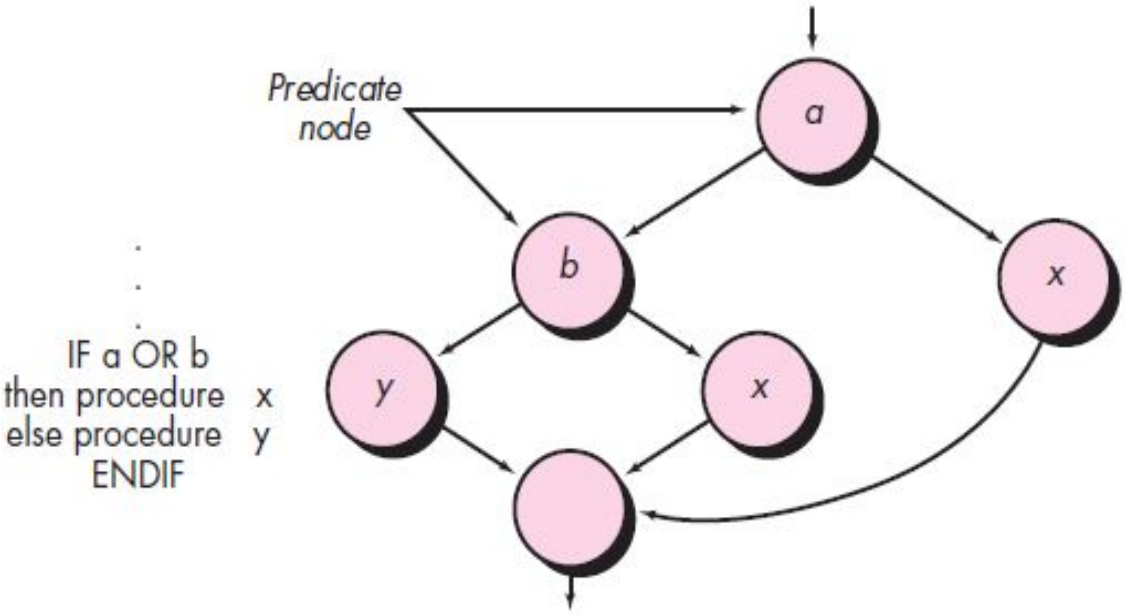




(a) Flowchart and (b) flow graph

- Consider the procedural design representation in **Figure (a)**. Here, a flowchart is used to depict program control structure. **Figure (b)** maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).

- Referring to Figure (b), each circle, called a ***flow graph node,*** represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called ***edges or links,*** represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called ***regions***. When counting regions, we include the area outside the graph as a region.



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

Independent Program Paths

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

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

- ***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 (b), 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 (b) is 4.

```
PROCEDURE average;

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

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

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

PDL with nodes identified

Deriving Test Cases

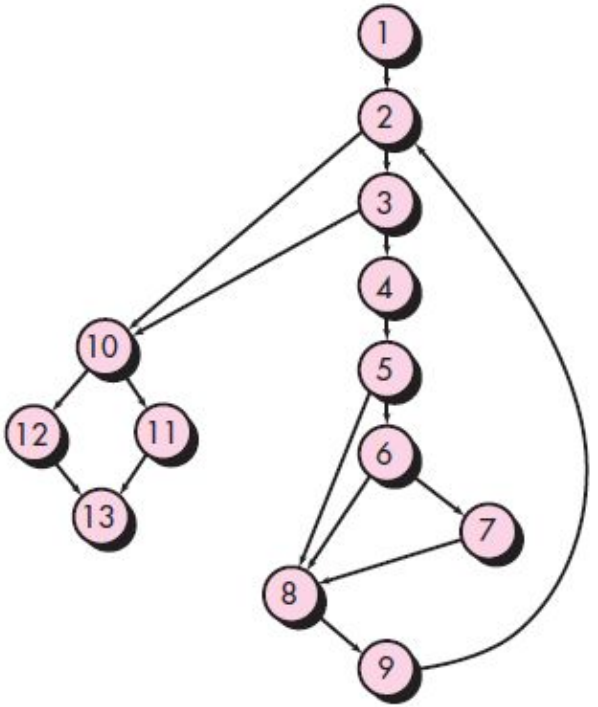
- The basis path testing method can be applied to a procedural design or to source code. The procedure *average,* depicted in PDL in Figure, will be used as an example to illustrate each step in the test-case design method. Note that average, although an extremely simple algorithm, contains compound 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. Referring to the PDL for *average in* Figure, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes.

2. Determine the cyclomatic complexity of the resultant flow graph.

The cyclomatic complexity *V(G)* is determined by applying the algorithms. 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,

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



Flow graph for the procedure *average*

3. Determine a basis set of linearly independent paths.

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

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

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

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-. . .

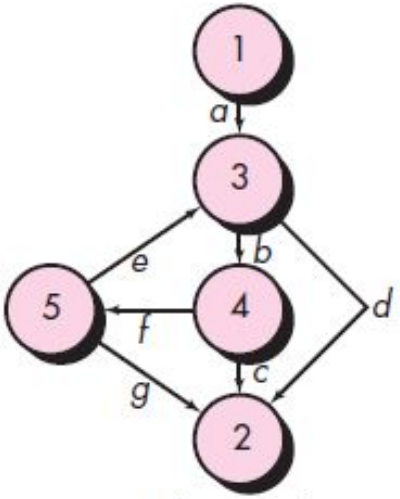
The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through

the remainder of the control structure is acceptable.

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.

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.



Flow graph

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

Graph matrix

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

CONTROL STRUCTURE TESTING

- Although basis path testing is simple and highly effective, it is not sufficient in itself. These broaden testing coverage and improve the quality of white-box testing.

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), , or . A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (), AND (&), 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.

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 the definition of X in statement S is live at statement S'.*

- One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy. It has been shown that DU testing does not guarantee the coverage of all branches of a program.

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*.

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.*

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.

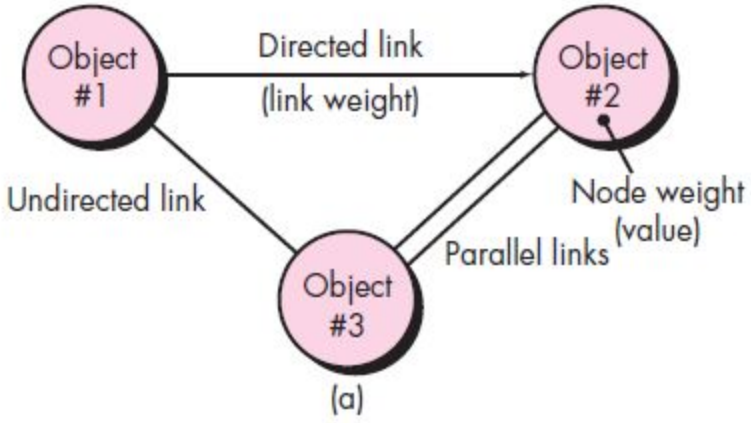
Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

BLACK-BOX TESTING

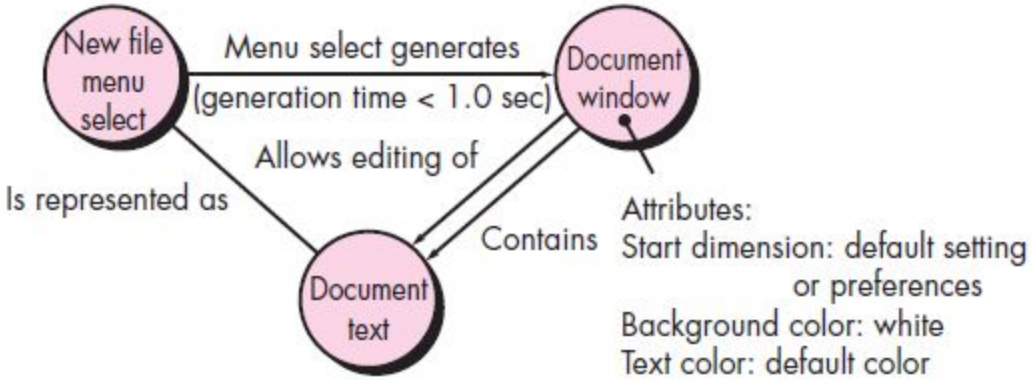
- *Black-box testing,* also called *behavioral testing,* focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white box methods.
- A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.
- *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**.
- 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.

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



(a)



(b)

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

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. 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.

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.

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.

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.

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**.

Alpha and Beta Testing

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

*The **beta test** is conducted at one or more end-user sites.* Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

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

SYSTEM TESTING

- 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.

- You should anticipate potential interfacing problems and **(1) design error-handling paths that test all information coming from other elements of the system**, **(2) conduct a series of tests that simulate bad data or other potential errors at the software interface**, **(3) record the results of tests to use as "evidence" if finger pointing does occur**, and **(4) participate in planning and design of system tests to ensure that software is adequately tested**.

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.

Security Testing

- Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer: “The system’s security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack.”
- During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! **The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to break down any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.**
- Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

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.

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. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

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.

DEBUGGING

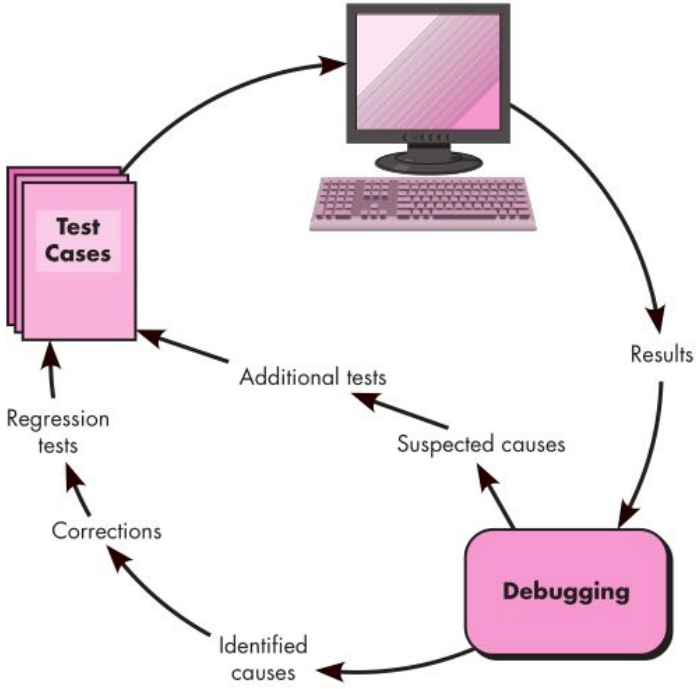
- ***Debugging* occurs as a consequence of successful testing**. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. As a software engineer, you are often confronted with a "symptomatic" indication of a software problem as you evaluate the results of a test. That is, the external manifestation of the error and its internal cause may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

The Debugging Process

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

- The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found.

- In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.



The debugging process

Debugging Strategies

- Regardless of the approach that is taken, debugging has one overriding objective-to find and correct the cause of a software error or defect. The objective is realized by a combination of systematic evaluation, intuition, and luck.
- 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.
- ***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.

Automated debugging.

- Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted.
- 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.

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!"

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.

SOFTWARE MAINTENANCE

- It begins almost immediately.
- Software is released to end users, and within days, bug reports filter back to the software engineering organization.
- Within weeks, one class of users indicates that the software must be changed so that it can accommodate the special needs of their environment.
- And within months, another corporate group who wanted nothing to do with the software when it was released now recognizes that it may provide them with unexpected benefit. They'll need a few enhancements to make it work in their world.
- **The challenge of software maintenance has begun.**
- You're faced with a growing queue of bug fixes, adaptation requests, and outright enhancements that must be planned, scheduled, and ultimately accomplished. Before long, the queue has grown long and the work it implies threatens to overwhelm the available resources.
- As time passes, your organization finds that it's spending more money and time maintaining existing programs than it is engineering new applications.
- In fact, it's not unusual for a software organization to expend as much as 60 to 70 percent of all resources on software maintenance.

- Another reason for the software maintenance problem is the mobility of software people. It is likely that the software team (or person) that did the original work is no longer around. Worse, other generations of software people have modified the system and moved on. And today, there may be no one left who has any direct knowledge of the legacy system.

Maintainability

- *Maintainability* is a qualitative indication of the ease with which existing software can be corrected, adapted, or enhanced. Much of what software engineering is about is building systems that exhibit high maintainability.
- Maintainable software exhibits effective modularity.
- It makes use of design patterns that allow ease of understanding. It has been constructed using well-defined coding standards and conventions, leading to source code that is self-documenting and understandable.
- It has undergone a variety of quality assurance techniques that have uncovered potential maintenance problems before the software is released. It has been created by software engineers who recognize that they may not be around when changes must be made.
- The design and implementation of the software must “assist” the person who is making the change.

SOFTWARE REENGINEERING

- Reengineering occurs at two different levels of abstraction. At the business level, reengineering focuses on the business process with the intent of making changes to improve competitiveness in some area of the business. At the software level, reengineering examines information systems and applications with the intent of restructuring or reconstructing them so that they exhibit higher quality.

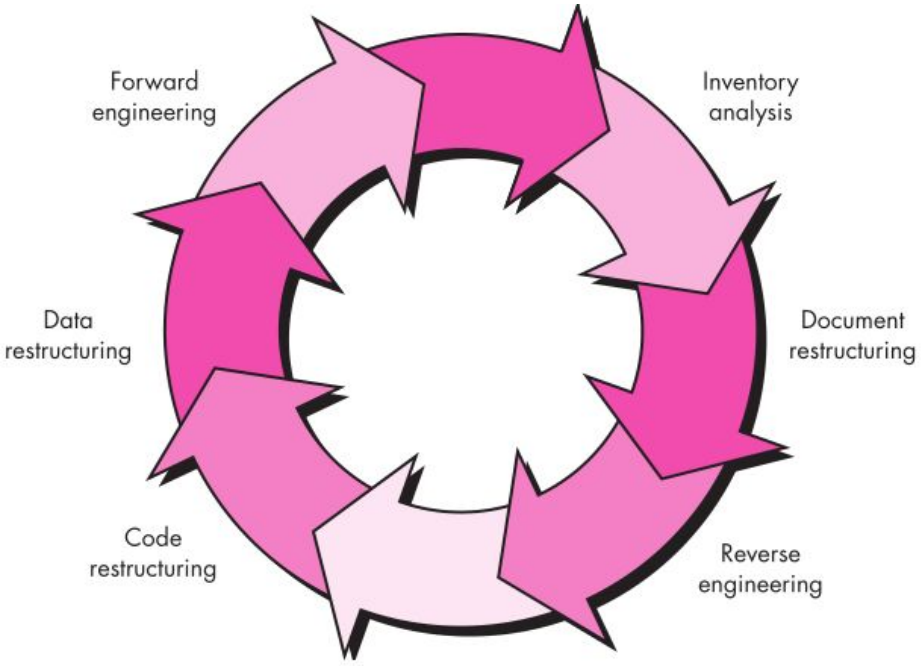
A Software Reengineering Process Model

- Reengineering takes time, it costs significant amounts of money, and it absorbs resources that might be otherwise occupied on immediate concerns. For all of these reasons, reengineering is not accomplished in a few months or even a few years.
- A workable strategy is encompassed in a reengineering process model.
- Reengineering is a rebuilding activity.

- To better understand reengineering, consider an analogous activity: the rebuilding of a house. Consider the following situation. You've purchased a house in another state. You've never actually seen the property, but you acquired it at an amazingly low price, with the warning that it might have to be completely rebuilt.

How would you proceed?

- Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you (or a professional inspector) would create a list of criteria so that your inspection would be systematic.
- Before you tear down and rebuild the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding (at much lower cost and in much less time).
- Before you start rebuilding be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.
- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality—today and in the future.



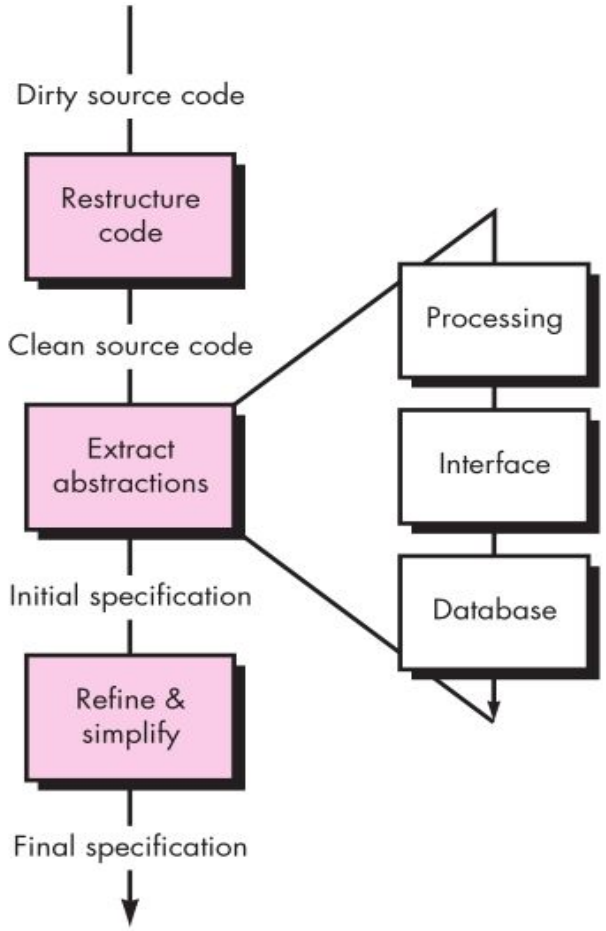
A software reengineering process model

Software Reengineering Activities

1. Inventory analysis.
2. Document restructuring.
3. Reverse engineering.
4. Code restructuring.
5. Data restructuring.
6. Forward engineering.

REVERSE ENGINEERING

- Reverse engineering can extract design information from source code, but the abstraction level, the completeness of the documentation, the degree to which tools and a human analyst work together, and the directionality of the process are highly variable.
- The *abstraction level* of a reverse engineering process and the tools used to effect it refers to the sophistication of the design information that can be extracted from source code.
- The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases.
- Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is “integrated” with automated tools to create an effective reverse engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.
- If the *directionality* of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.



The reverse engineering process

The core of reverse engineering is an activity called *extract abstractions*. You must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.