UNIT 4

Chapter 17

Software Testing Strategies

- No matter how carefully you analyze, design and develop software, errors are bound to be there and hence testing of the software before you deliver it to the customer is absolutely necessary.
- Proper testing enables you to remove most of the errors and improve the quality of software delivered to the customer.
- But there are number of different techniques available for testing of software. We also have questions like how much time, resources and effort should be spent on testing, which techniques should be used, when should the planning take place, when should the actual testing take place.
- These are the things that we decide when we develop testing strategy.
- There are number of different strategies proposed in the literature but all have certain **common characteristics**. They are

- **Common characteristics in Software Strategy:**
  - FTRs should be performed before testing starts. FTRs remove errors at the earlier stage and hence eliminate most of the errors before testing starts.
  - Testing begins at component level and works “outward” toward the integration of the entire computer based system.
  - Different testing techniques are appropriate at different points of time.
  - Testing is conducted by the developer of the software and an independent test group.
  - Testing (finding errors) and debugging (removing errors) are different activities, but debugging must be accommodated in any testing strategy.

A strategic approach to s/w testing

1) **Verification and validation**

- Software testing is one element of a broader topic that is often referred to as verification and validation.
- Other elements in V and V are other SQA activities like FTR, quality and configuration audit, performance monitoring, simulation, feasibility study, documentation review etc.
- <u>Verification</u> refers to the set of activities that ensure that software <u>correctly implements a specific function</u> (whether it is required or not is a different issue).
- <u>Validation</u> refers to a different set of activities that ensure that the software that has been built is <u>traceable</u> to <u>customer requirements.</u>
- Verification – "Are we building the product right?"
- Validation - "Are we building the right product?"
  - Are we conducting the MCA course properly? Does market need MCAs? Validation deals with the requirements.
  - Are we teaching the subject properly? Are we teaching the proper subject? COBOL, .Net

2) Organizing for s/w testing

- For the developer analysis, design and coding seem to be constructive activities (creating software) whereas testing seems to be psychologically a destructive activity (since you are trying to find errors or break the software).
- The developer also would like to show that the software works and there are no errors and hence he/she would be biased at the time of testing.
- So there is a need for independent testing.
- But the developer should test the individual components and also should be available to correct the errors.
- So developer also has a role to play in testing.
- Similarly the testing group also has to be involved at the specification stage and planning of test procedures. This group is called independent test group (ITG).

developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

independent tester

Must learn about the system,

but, will attempt to break it

and, is driven by quality

- **3)Software Testing strategy**
- In software engineering we are carrying out four major activities namely system engineering, analysis, design and coding.
- Testing also has to be done for each of these levels. The relationship is as shown below:

Activity	Testing Step	Type of testing
System Engg.	System Testing	
Analysis	Validation	Black-box
Design	Integration testing	Black-box
Coding	Unit testing	White-box

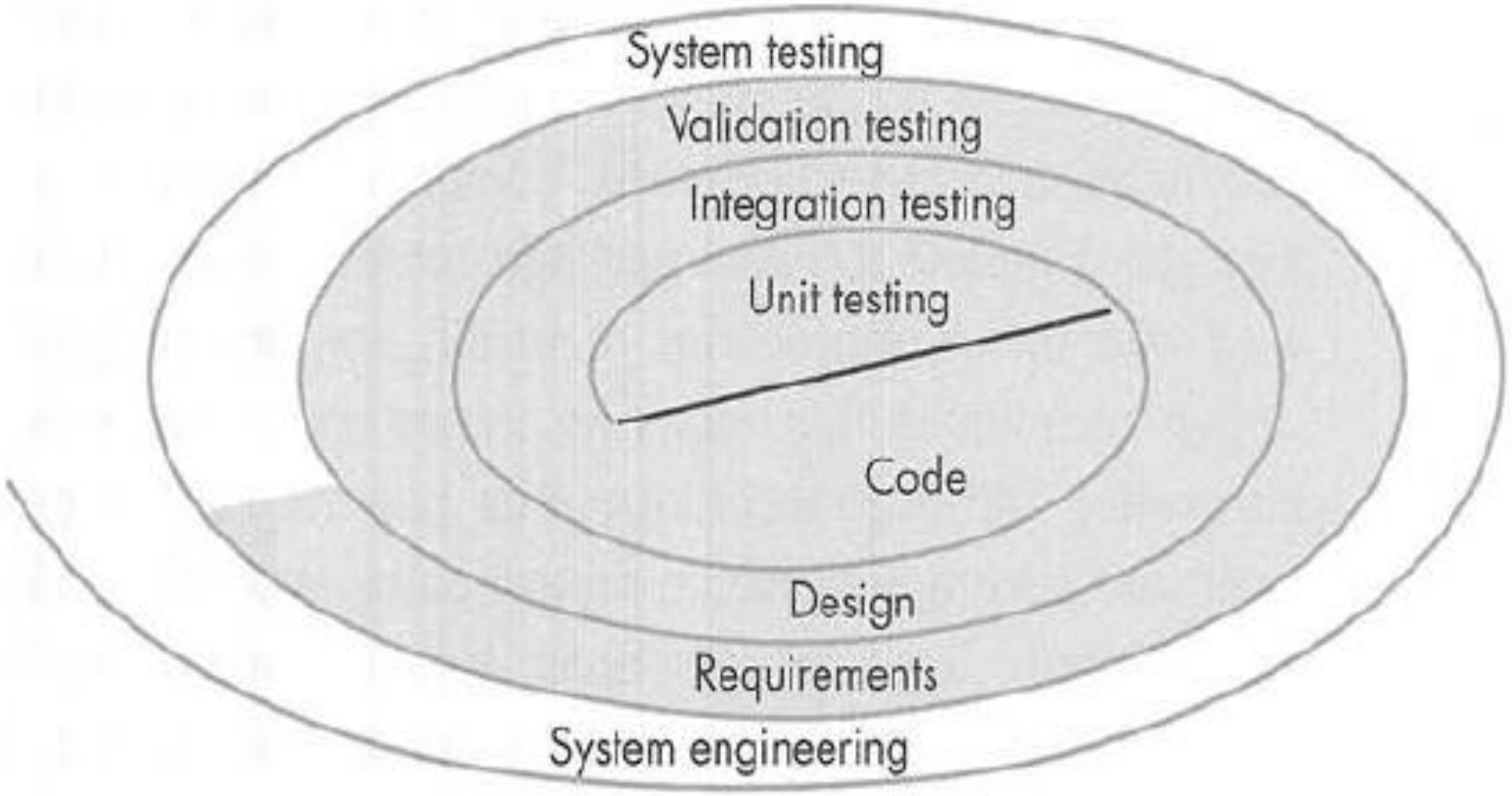
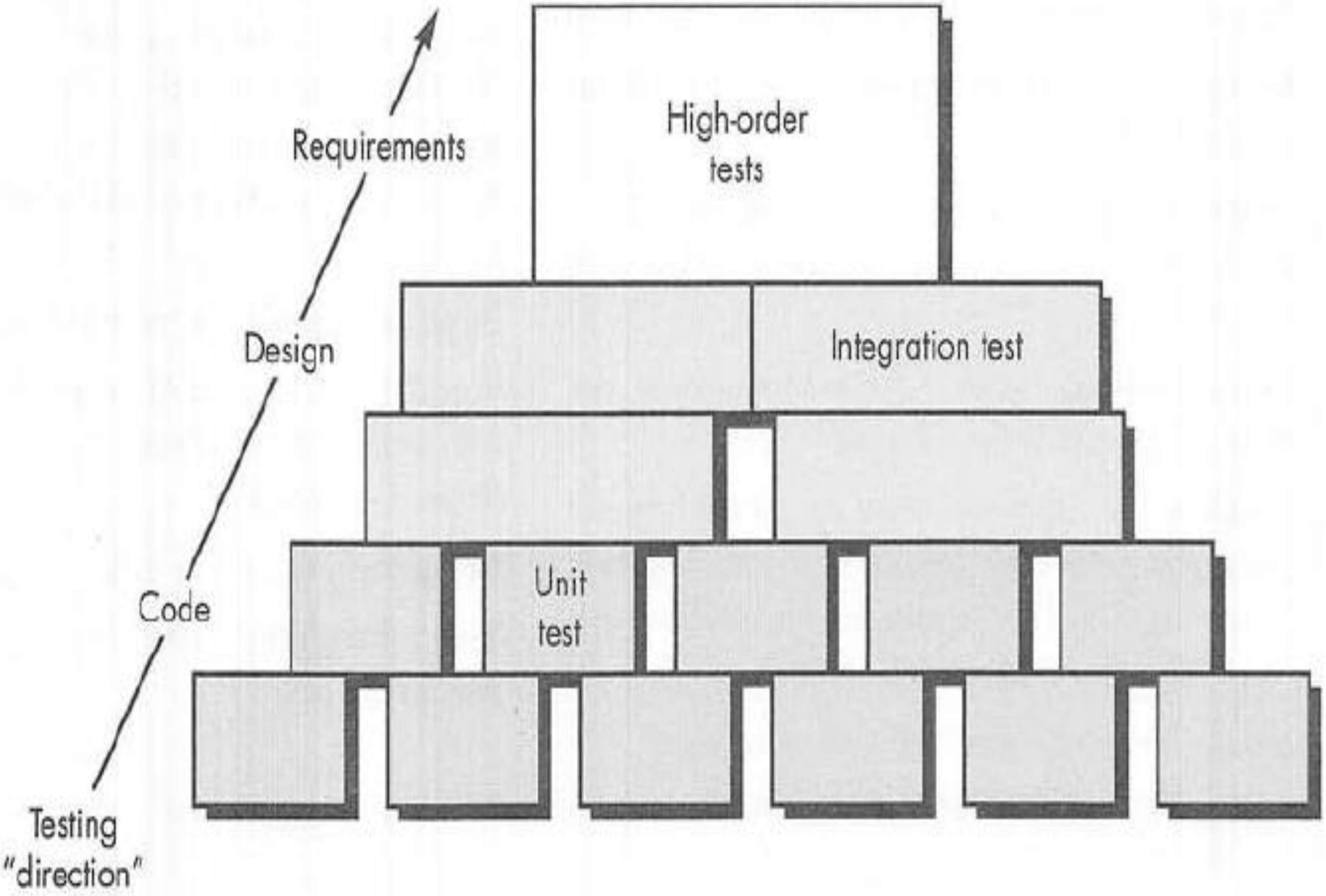

System testing
Validation testing
Integration testing
Unit testing
Code
Design
Requirements
System engineering

Fig: s/w testing steps



- Unit testing: component properly functions as a unit, ensure complete coverage and maximum error detection.
- Integration testing: dual problems of verification and program construction, focus on i/p and o/p are more prevalent during integration.
- High-order tests: validation testing provide final assurance that s/w meets all informational, functional, behavioral, and performance requirements.
- Once validated combined with other system elements.

4)Criteria for completion of testing

- Whenever software is being developed and the time comes to do testing, there is a question that invariably arises and that is how much testing should be done? Or when do you stop testing? There is no definitive (best, perfect) answer. Different people have come up with different answers and suggestions. Some of them are :
  - Testing never stops since while the end user uses the system it is being tested.
  - You are done with testing when you run out of time or money.
  - Then there is statistical criteria involving probability and confidence.

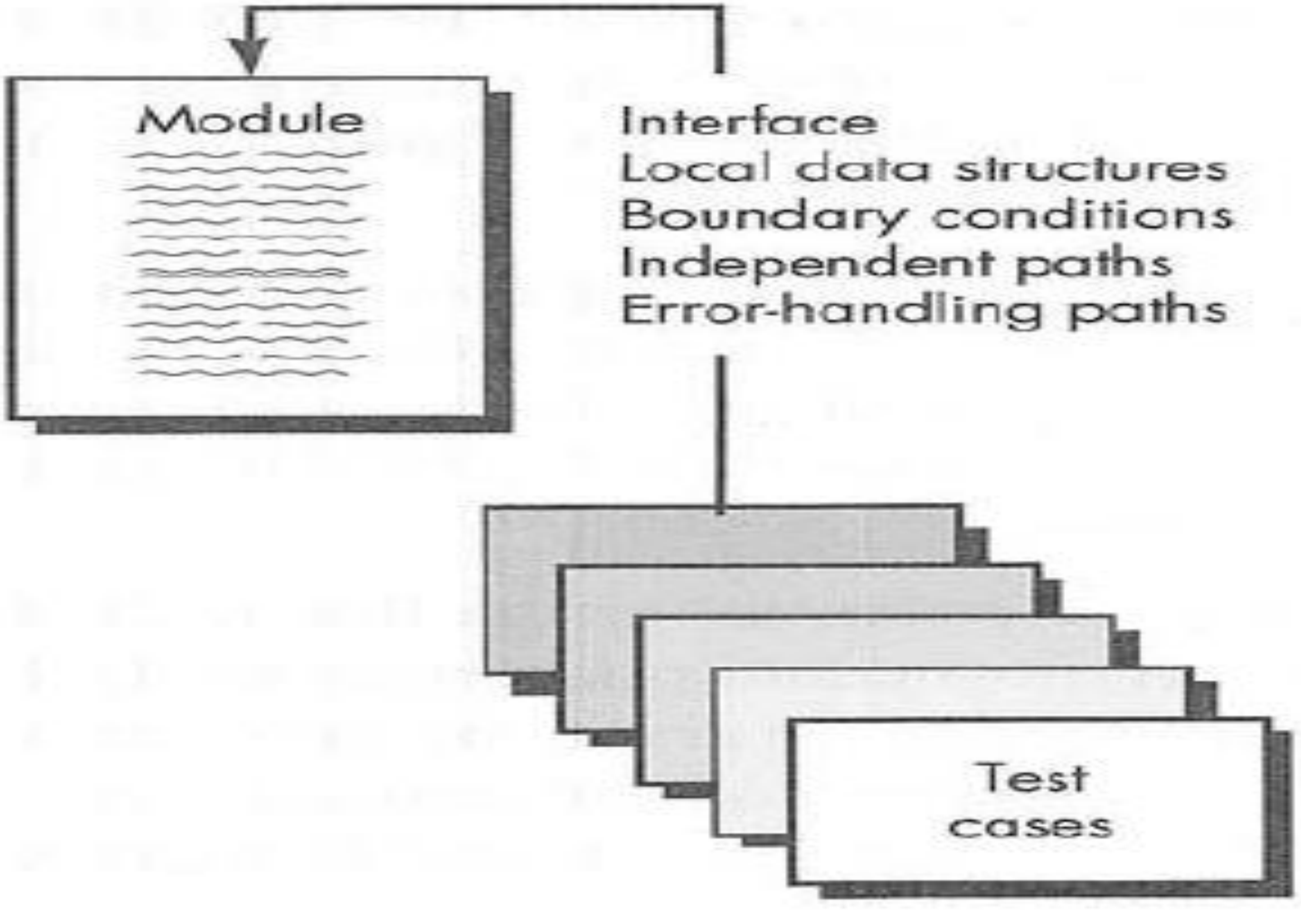
Test strategies for conventional s/w

- There are many strategies for conventional software.
- At one extreme is the strategy of waiting till the whole software is ready and then test it for all the errors at one stroke.
- At the other extreme is the strategy of testing daily. None of these approaches is practical and hence one can take incremental view of testing.
- There are following four phases or Levels:
  - Unit testing (individual programs),
  - integration testing (modules),
  - validation testing (whole software) and
  - system testing (the whole product).

Unit Testing:

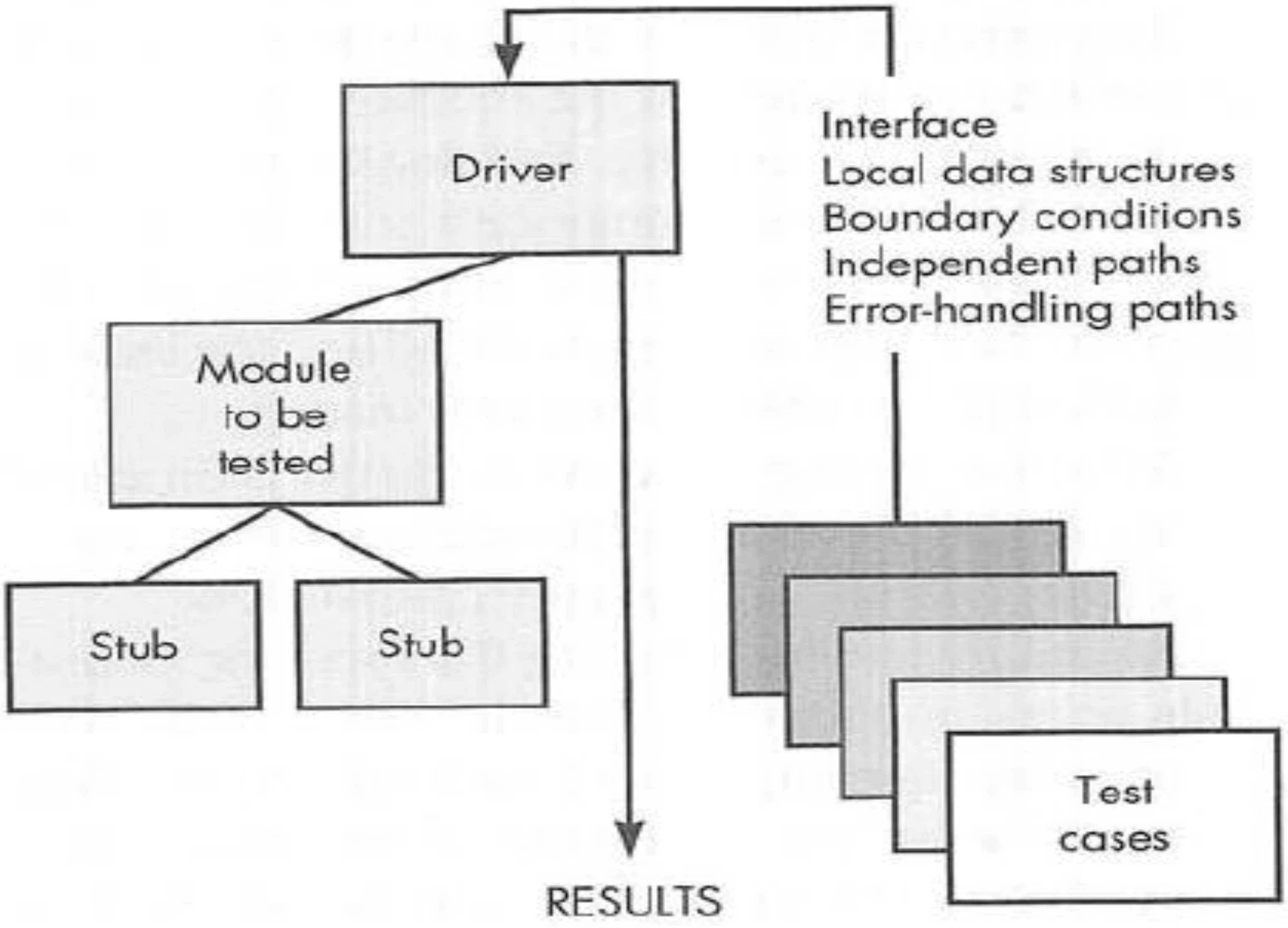
- **Unit Testing** is a level of the software testing process where individual units/components of a software/system are tested.
- The purpose is to validate that each unit of the software performs as designed.
- Unit testing focuses verification effort on the smallest unit of software design – the software component.
- It is a white-box testing method i.e. it focuses on the internal processing logic and data structures of a component or unit.
- Using the component-level design description as a guide, important control paths are tested to uncover errors.
- This type of testing can be conducted in parallel for multiple components.
- A unit is the smallest testable part of software. It usually has one or a few inputs and usually a single output.

- In procedural programming a unit may be an individual program, function, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.
- **Who performs it?**
  - Unit Testing is normally performed by software developers themselves or their peers. In rare cases it may also be performed by independent software testers.
- **Unit-test considerations:**
  - Following five different types of tests are conducted :
    - Interface : To ensure that information properly flows into and out of the program unit under test.
    - Local data structure : Data stored temporarily maintains its integrity
    - Boundary conditions : Program operates properly at boundary conditions
    - Independent paths: tested to ensure that all statements get executed at least once. Cyclomatic complexity
    - Error handling paths: are tested to make sure that they work properly.


Module
Interface
Local data structures
Boundary conditions
Independent paths
Error-handling paths
Test cases

- Unit test procedures
  - Unit testing starts as soon as coding of that part is over.
  - Now the program that we are interested in testing may call some other programs(subordinate) and would definitely be called by a main program.
  - Since these programs may not be ready we have to develop dummy programs so that this program can be tested.
  - The <u>dummy</u> programs are called <u>drivers</u> (the calling program) and <u>stubs</u> (the called programs).
  - Drivers and stubs are <u>overheads</u> since they are <u>not delivered</u> with the software. They are developed only for testing.
  - The design of unit tests can occur before coding begins or after source code has been generated.
  - Each test case should be coupled with a set of expected results.
  - Driver and/or stub s/w must often be developed for each unit test.
  - Stubs serve to replace modules that are subordinate the component to be tested.
  - If drivers and stubs are kept relatively simple, overhead can be low, but when not possible; complete testing can be postponed until the integration test step.

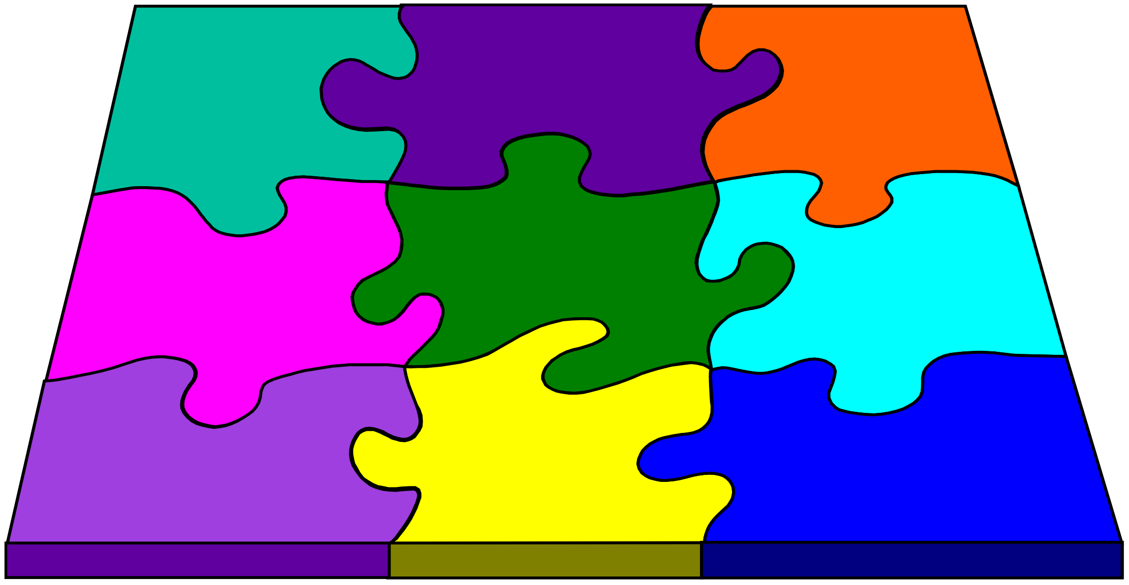
Fig: unit test environment



Integration Testing

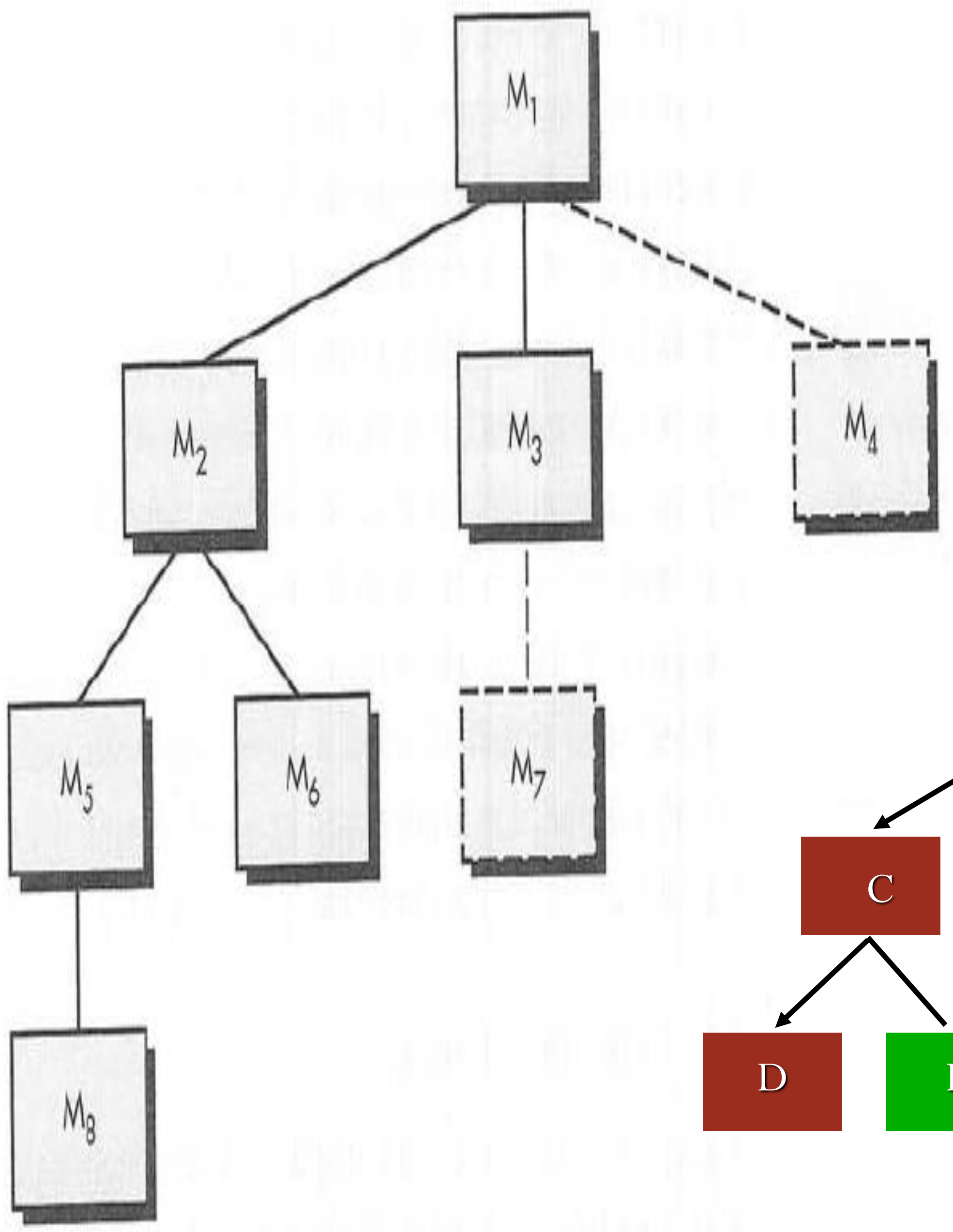
- Once parts work properly we have to integrate (link) them to get the whole system.
- But there is no guarantee that the integrated system will work properly since parts are working properly.
- That is why we need integration testing.
- It is a systematic technique for constructing the program structure (as per design) while at the same time conducting tests to uncover errors associated with interfacing.
- For integration there are two approaches:
  - "big bang" (or non-incremental) and
  - incremental.
- In big bang approach all components are combined and then the whole system is tested as a whole. This approach is not advisable as finding of errors and correction is more difficult.

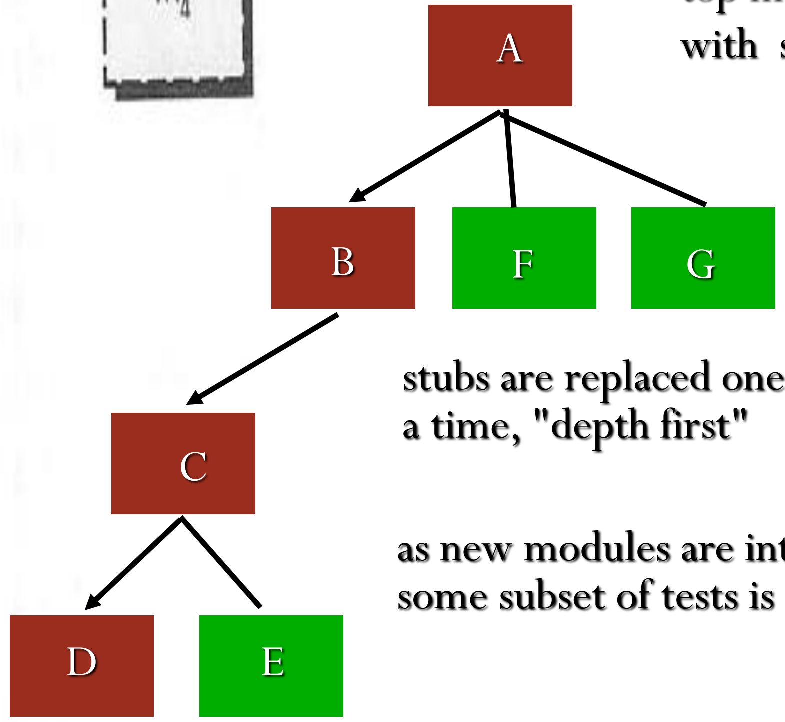
Big-Bang Approach



- In incremental approach the program is constructed and tested in <u>small increments</u>, where errors are easier to isolate and correct and interfaces are tested more completely.
- There are number of different incremental integration strategies like
  - top-down integration,
  - bottom-up integration,
  - regression testing and
  - smoke testing.

- <u>Top-down integration:</u>
- It is an incremental approach to construction of program structure.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.
- The integration process is performed in following 5 steps:
  1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
  2. Depending on the integration approach (depth or breadth first) subordinate stubs are replaced one at a time with actual components.
  3. Tests are conducted as each component is integrated.
  4. On completion of each set of tests, another stub is replaced with the real component.
  5. Regression testing may be conducted to ensure that new errors have not been introduced.





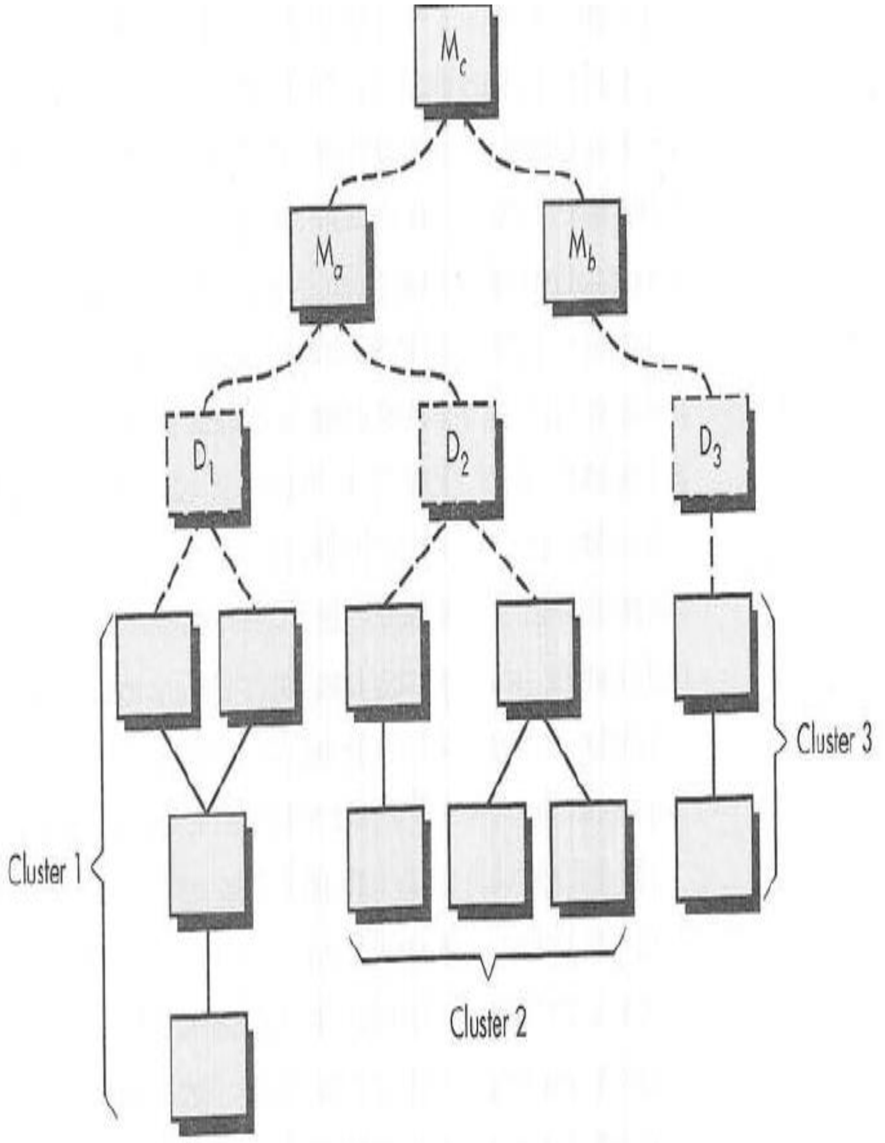
top module is tested with stubs

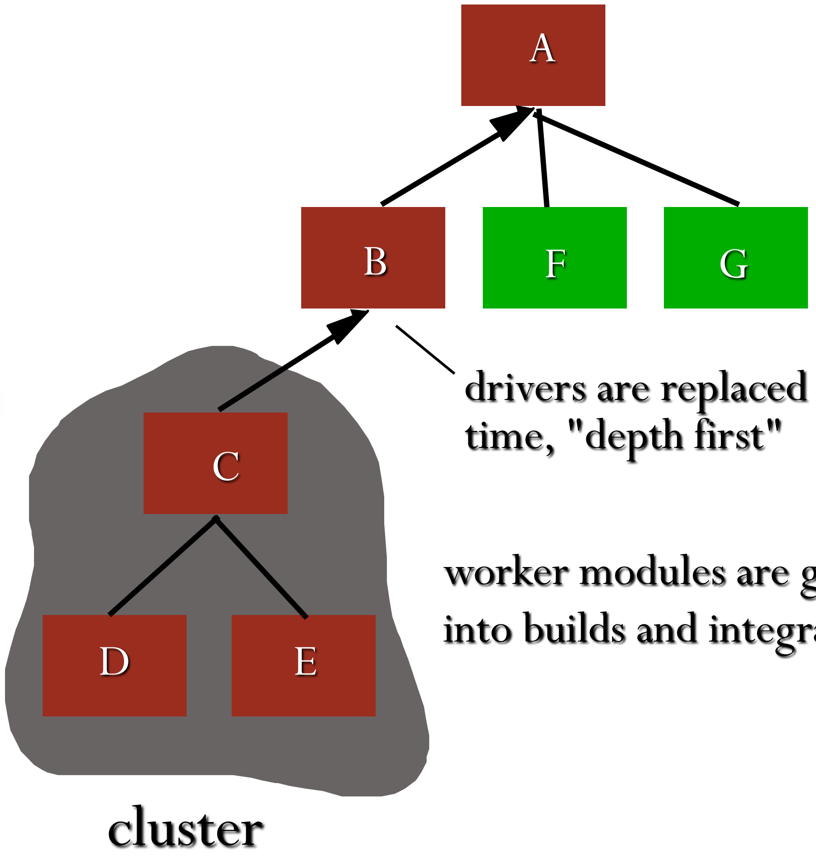
stubs are replaced one at a time, "depth first"

as new modules are integrated, some subset of tests is re-run

- Since decision making occurs at upper levels the <u>control structure is tested early.</u>
- If depth first integration is selected, a complete function may be implemented and demonstrated. Financial Accounting, sales, purchase, inventory.
- There are some problems with this approach.
- The most common is when data from the lower levels is required for testing of upper level.
- But that data is not available since we are using stubs at lower levels (not the actual components) and replacing them one by one with real components.
- There are 3 choices :
  - delay testing of some functionality till stubs are replaced with actual components (then what is the use of top-down approach?),
  - develop stubs with limited functionality for testing (overhead) and
  - use bottom-up approach.

- <u>Bottom-up integration</u>
- Start integration and testing from bottom up. There is no need of stubs since actual components are already tested. The steps are as follows:
- Low-level components are combined into <u>clusters</u> (also called <u>builds</u>) that perform a specific software subfunction.
- A driver 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.





drivers are replaced one at a time, "depth first"

worker modules are grouped into builds and integrated

- Regression testing:
- Each time a new module is added as part of integration testing, the software changes.
- New data flow paths are established, new I/O may occur, and new control logic is invoked, errors are corrected.
- These changes may cause problems with functions that previously worked flawlessly.
- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated any unintended side effects. Medicine.

- <u>Smoke Testing:</u>
- It can be considered as <u>rolling</u> integration.
- Basically you test the whole product daily for the major functionalities.
- It is used for "shrink-wrapped" software products.
- It is used as a pacing mechanism for time critical projects.
- Encompasses following activities:
  - s/w components that have been translated into code are integrated into a build. A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its functions.
  - The build is integrated with other builds and the entire product is tested daily.

- Benefits:
  - Integration risk is minimized.
  - Quality of end product is improved.
  - Error diagnosis and correction are simplified.
  - Progress is easier to assess.

Validation testing

- This testing focuses on user visible action and user- recognizable output from the system
- SRS describes all user-visible attributes of the s/w and contains a validation criteria section that forms the basis for a validation testing approach

1) <u>Validation test criteria</u>

- A test plan outlines the classes of test to be conducted, and a test procedure defines specific test cases that are designed to ensure transportability, compatibility, error recovery, maintainability.
- After validation test two possible conditions exist:

  - the function conforms to specification and is accepted

  - a deviation from specification is uncovered and deficiency list is created

2) Configuration review

- Intent is to ensure that all elements of the s/w configuration have been properly developed, are cataloged
- Sometimes called audit.(ch:20)

3) Alpha and beta testing

- It is difficult to predict that how customer will use a program
- Alpha and beta testing are used to uncover errors that only the end user seems able to find, and in a situation when product will be used by many customers.

- Alpha test:

  - is conducted at the developer's site by a representative group of end users.

  - the s/w is used in a natural setting with the developer "looking over the shoulder" of the users and recoding errors and usage problems.

- Beta test:

  - it is conducted at one or more end-user site

  - developer is not present so it is a live application of s/w in an environment that can not be controlled by developer

  - customer records all problems and reports to developer at the regular intervals

  - modifications are done and prepare for the release to the entire customer base

- A variation in beta testing called *customer acceptance testing*
- In which custom s/w is delivered to a customer under contract
- customer performs a series of specific tests in an attempt to uncover error before accepting the s/w from the developer

System testing

- It is a series of different tests whose primary purpose is to fully exercise the computer-based system.
- These tests fall outside the scope of the software process and are not conducted solely by software engineers.
- Types of system testing:

1) Recovery testing:

- Many computer-based systems must recover from faults and resume processing with little or no downtime.
- *It is system test that forces the s/w to fail in a variety of ways and verifies that recovery is properly performed*
- If recovery is automatic- reinitializing, checkpoint mechanisms, data recovery, and restart are evaluated
- If recovery requires human interaction- MTTR is evaluated.

2) <u>Security testing</u>:

- It attempts to verify that protection mechanisms built into a system, protect it from improper access or entry in the system.
- The tester plays a role of the individual who desires to access the system.
- He may attempt to acquire password ,may attack the system with custom s/w designed to break down any defenses
- He may purposely cause system errors, hoping to penetrate during recovery.
- Good security testing will ultimately access a system.
- Role of the designer is to make access cost more than the value of the information that will be obtained.

3) Stress testing:

- Stress tests are designed to confront programs with abnormal situations
- Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency and volume
  - special tests may be designed that generate ten interrupts per second, when one or two is the average rate
  - input data rates may be increased by an order of magnitude to determine how input functions will respond
  - test cases that require maximum memory or other resources are executed,
  - test cases that may cause excessive hunting for disk-resident data are created.
- Variation in stress testing is called *sensitivity testing*
- sometimes a very small range of data contained within the bounds of valid or data for program may cause extreme or erroneous processing
- Sensitivity testing attempts to uncover data combinations within valid i/p classes that may cause improper processing

4) Performance testing:

- Performance testing is designed to test the run time performance of s/w 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
- It is often coupled with stress testing and usually require both h/w and s/w instrumentation
- It is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur,
- and sample machine states on a regular basis.
- By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

5) Deployment Testing:

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

Chapter 18

TESTING CONVENTIONAL APPLICATIONS

SOFTWARE TESTING FUNDAMENTALS

- The goal of testing is to find errors, and a good test is one that has a high probability of finding an error.
- Therefore, you should design and implement a computer based system or a product with "testability" in mind.
- s/w testability is simply how easily a computer program can be tested .
- **Characteristics of good tests:**
  - *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 be neither too simple nor too complex.*

Black Box Testing

- Also known as functional testing and behavioral testing, Black-box testing focuses on the functional requirements of the software.
- The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure.
- **Black Box Testing** is testing without knowledge of the internal workings of the item being tested.
- For example, when black box testing is applied to software engineering, the tester would only know the "legal" inputs and what the expected outputs should be, but not how the program actually arrives at those outputs.

- The tester does not ever examine the programming code.
- The types of testing under this strategy are totally based/focused on the **testing for requirements and functionality of the work product/software application**.
- Black box testing is sometimes also called as "Opaque Testing", "Functional/Behavioral Testing" and "Closed Box Testing".
- It attempts to find errors in following categories:
  - incorrect or missing functions,
  - interface errors,
  - errors in data structures or
  - external database access

Equivalence Partitioning

- Divides input domain of program into classes of data from which test cases can be derived.
- Goals:-

1) To reduce the number of test cases to a necessary minimum.
2) To select the right test cases to cover all possible scenarios.

- Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for input condition.
- An <u>equivalence class</u> represents a set of valid or invalid states for input condition.
- 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 classes are defined.
4. If input condition is Boolean. One valid and one invalid classes are defined.

- **Example :** An input has certain ranges, one valid and two invalid equivalence classes are defined. This may be best explained at the following example of a function which has the pass parameter "month" of a date. The valid range for the month is 1 to 12, standing for January to December. **This valid range is called a partition. In this example there are two further partitions of invalid ranges**. The first invalid partition would be <= 0 and the second invalid partition would be >= 13.

- …. ..-2 -1 0 1 ………….. 12 13 14 15 …..
- invalid partition 1   valid partition   invalid partition 2
- (one valid and two invalid equivalence classes are defined)

BVA (Boundary Value Analysis)

- The boundaries of **software component input ranges** are areas of frequent problems.
- A greater no of errors occurs at the boundaries of input domain rather than in the “center”.
- BVA leads to selection of test cases that exercise bounding values.
- if (month > 0 && month < 13)
- But a common programming error may check a wrong range e.g. starting the range at 0 by writing:
- if (month >= 0 && month < 13)
- For more complex range checks in a program this may be a problem which is not so easily spotted as in the above simple example.
- To set up boundary value analysis test cases you first have to determine which boundaries you have at the interface of a software component. This has to be done by applying the equivalence partitioning technique. Boundary value analysis and equivalence partitioning are inevitably linked together.

- …. ..-2 -1 0 1 ………….. 12 13 14 15 …..
- invalid partition 1   valid partition   invalid partition 2

- Applying boundary value analysis you have to select now a test case at each side of the boundary between two partitions. In the above example this would be 0 and 1 for the lower boundary as well as 12 and 13 for the upper boundary
- The boundary value analysis can have 6 text cases. n, n-1,n+1 for the upper limit and n, n-1,n+1 for the lower limit.

White-box testing

- White-box testing, also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing.
- 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.

- Why we perform WBT?
  - **To ensure:**
  - That all independent paths within a module have been exercised at least once.
  - All logical decisions verified on their true and false values.
  - All loops executed at their boundaries and within their operational bounds internal data structures validity.
- **To discover the following types of bugs:**
  - Logical error tend to creep into our work when we design and implement functions, conditions or controls that are out of the program
  - The design errors due to difference between logical flow of the program and the actual implementation
  - Typographical errors and syntax checking

- **3 Main White Box Testing Techniques:**
  - Statement Coverage
  - Branch Coverage
  - Path Coverage

1) Statement coverage:

- In a programming language, a statement is nothing but the line of code or instruction for the computer to understand and act accordingly. A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in a running mode.
- Hence "*Statement Coverage*", as the name itself suggests, it is the method of validating whether each and every line of the code is executed at least once.

2) Branch Coverage:

- "Branch" in a programming language is like the "IF statements". An IF statement has two branches: T**rue and False**.
- So in Branch coverage (also called Decision coverage), we validate whether each branch is executed at least once.
- **In case of an "IF statement", there will be two test conditions:**
  - One to validate the true branch and,
  - Other to validate the false branch.
- Hence, in theory, Branch Coverage is a testing method which is when executed ensures that each and every branch from each decision point is executed.

3) Path Coverage

- Path coverage tests all the paths of the program. This is a comprehensive technique which ensures that all the paths of the program are traversed at least once. Path Coverage is even more powerful than Branch coverage. This technique is useful for testing the complex programs.

```
INPUT A & B
C = A + B
IF C>100
    PRINT "ITS DONE"
```

- For ***Statement Coverage*** – we would only need one test case to check all the lines of the code.
- **That means:**
- If I consider *TestCase_01 to be (A=40 and B=70)*, then all the lines of code will be executed.
- **Now the question arises:**
  - Is that sufficient?
  - What if I consider my Test case as A=33 and B=45?

- Because Statement coverage will only cover the true side, for the pseudo code, only one test case would NOT be sufficient to test it. As a tester, we have to consider the negative cases as well.
- Hence for maximum coverage, we need to consider **"*Branch Coverage*"**, which will evaluate the "FALSE" conditions.

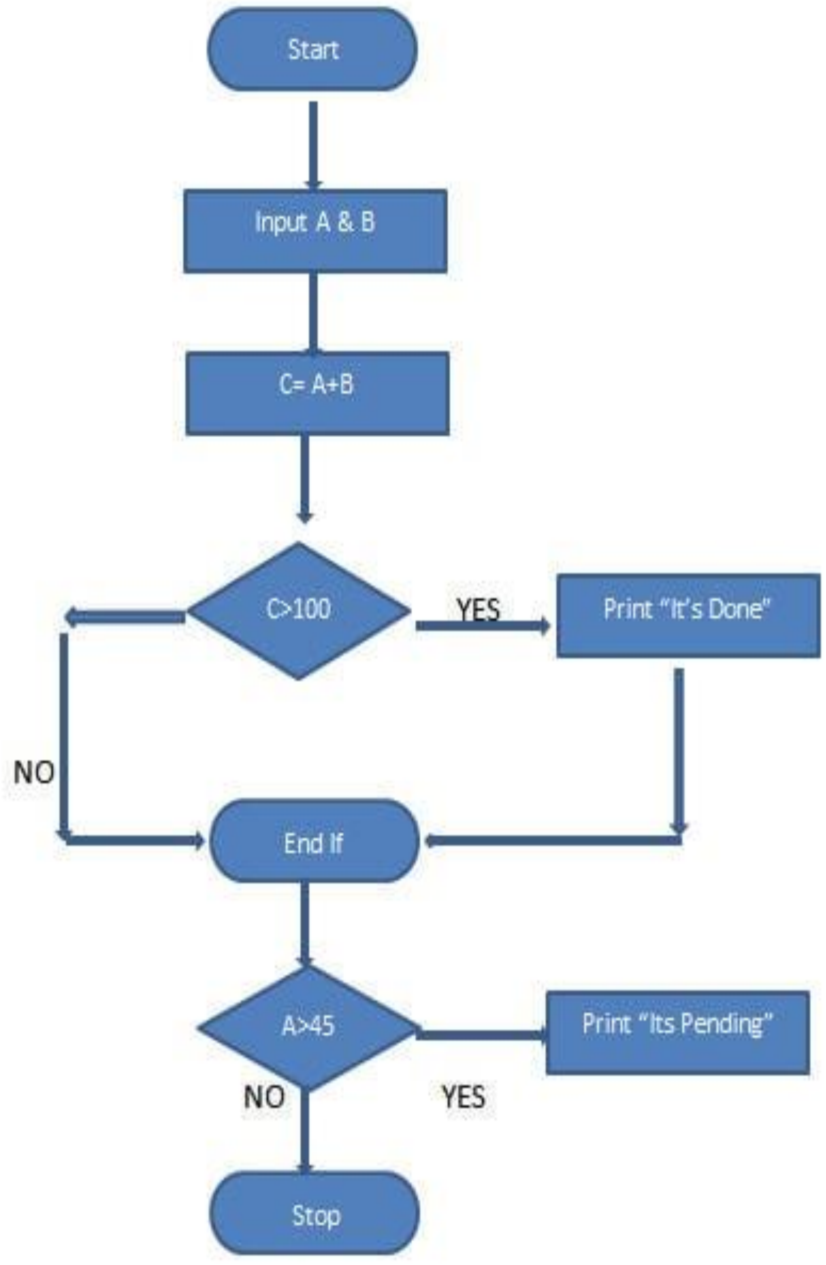
```
INPUT A & B
C = A + B
IF C>100
    PRINT "ITS DONE"
ELSE
    PRINT "ITS PENDING"
```

- for Branch coverage, we would require two test cases to complete the testing of this pseudo code.
- **TestCase_01**: A=57, B=45
- **TestCase_02**: A=25, B=30
- With this, we can see that each and every line of the code is executed at least once.

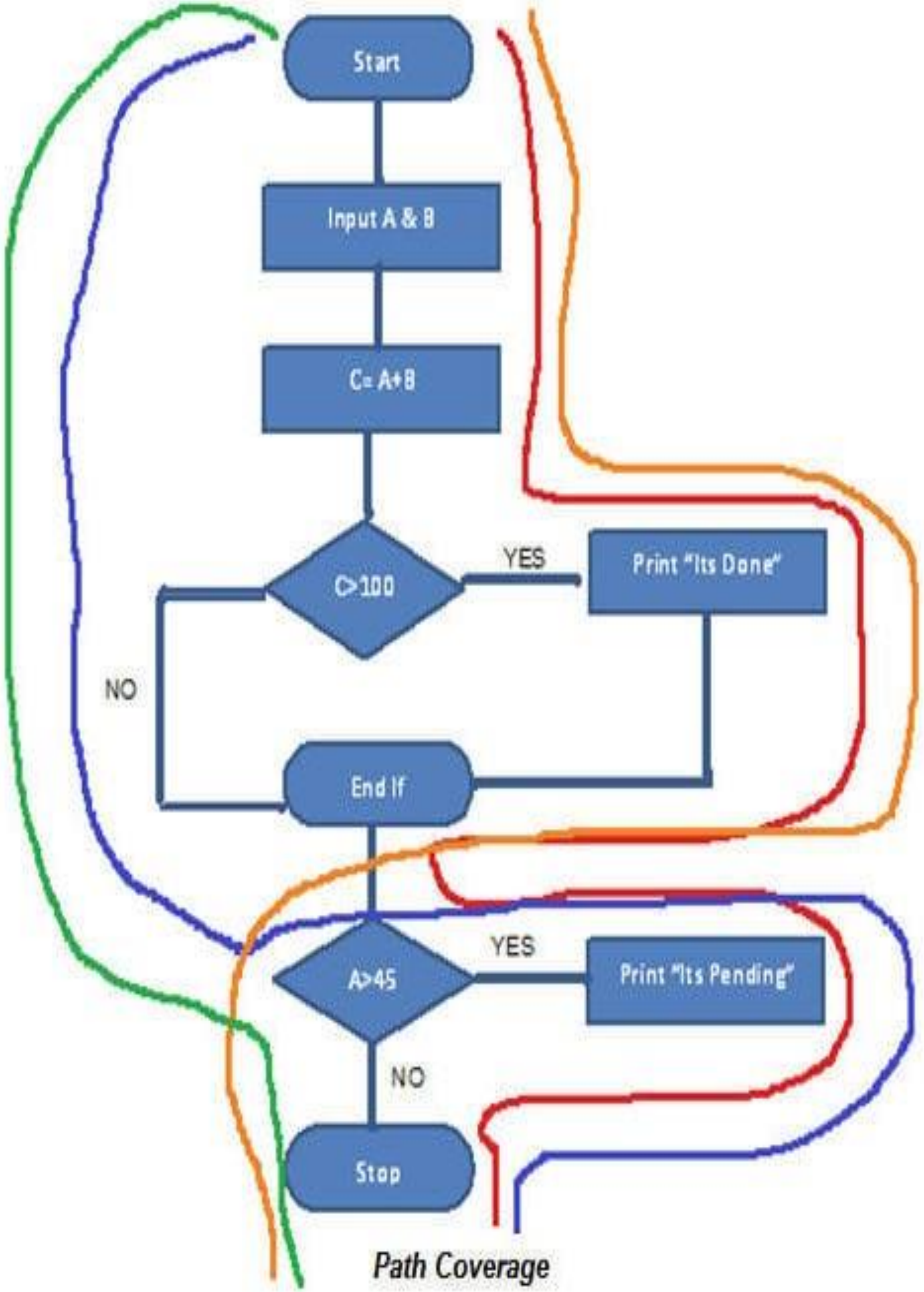
- Path coverage is used to test the complex code snippets, which basically involve loop statements or combination of loops and decision statements.

```
INPUT A & B
C = A + B
IF C>100
   PRINT “ITS DONE”
END IF
IF A>45
   PRINT “ITS PENDING”
END IF
```

- **In order to have the full coverage, we would need following test cases:**
  - **TestCase_01:** A=50, B=60
  - **TestCase_02**: A=55, B=40
  - **TestCase_03:** A=40, B=65
  - **TestCase_04:** A=30, B=30



Path Coverage



Path Coverage

Software Review

- Software reviews are a "filter" for the software process. That is, reviews are applied at various points during Software engineering and solve to uncover errors and defects that can then be removed.
- Software reviews "purify" soft-ware engineering Work products, including requirements and design models, code, and testing data.
- Technical work needs reviewing for the same reason that pencils need erasers: To err is human.
- The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape any one else.
- A review is a way of using the diversity of a group of people to: (Objectives)
  - Point out needed improvements in the product of a single person or team.
  - Confirm those parts of a product in which improvement is either not desired or not needed;
  - Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

- Many different types of reviews can be conducted as part of software engineering.
- Each has its place.
- An informal meeting around the coffee machine is a form of review, if technical problems are discussed.
- A formal presentation of software architecture to an audience of customers, management, and technical staff is also a form of review.
- A technical review (TR) is the most effective filter from a quality control standpoint.
- Conducted by software engineers(and others) for software engineers, the TR is an effective means for uncovering errors and improving software quality.

Static Testing Techniques

- **Informal Reviews:** This is one of the type of review which doesn't follow any process to find errors in the document. Under this technique , you just review the document and give informal comments on it.
- **Technical Reviews:** A team consisting of your peers, review the technical specification of the software product and checks whether it is suitable for the project. They try to find any discrepancies in the specifications and standards followed. This review concentrates mainly on the technical document related to the software such as Test Strategy, Test Plan and requirement specification documents.
- **Walkthrough:** The author of the work product explains the product to his team. Participants can ask questions if any. Meeting is led by the author. Scribe makes note of review comments

- **Inspection:** The main purpose is to find defects and meeting is led by trained moderator. This review is a formal type of review where it follows strict process to find the defects. Reviewers have checklist to review the work products .They record the defect and inform the participants to rectify those errors.
- **Static code Review:** This is systematic review of the software source code without executing the code. It checks the syntax of the code, coding standards, code optimization, etc. This review can be done at any point during development.

Informal Reviews:

- Informal reviews include a simple desk check of a software engineering work product with a colleague, a casual meeting (involving more than two people) for the purpose of reviewing a work product
- A simple desk check or a casual meeting conducted with a colleague is a review.
- However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches.
- But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.
- One way to improve the efficacy of a desk check review is to develop a set of simple review checklists for each major work product produced by the software team.
- The questions posed within the checklist are generic, but they will serve to guide the reviewers as they check the work product.

Formal Technical Review (FTR):

- A formal technical review (FTR) is a software quality control activity performed by software engineers (and others).
- A **software technical review** is a form of peer review in which "a team of qualified personnel ... examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards."
- Technical reviews may also provide recommendations of alternatives and examination of various alternatives.
- Technical review differs from software walkthroughs in its specific focus on the technical quality of the product reviewed.
- It differs from software inspection in its ability to suggest direct alterations to the product reviewed, and its lack of a direct focus on training and process improvement.
- The term formal technical review is sometimes used to mean a software inspection.

- IEEE 1028 recommends the inclusion of participants to fill the following roles:
  - The **Decision Maker** (the person for whom the technical review is conducted) determines if the review objectives have been met.
  - The **Review Leader** is responsible for performing administrative tasks relative to the review, ensuring orderly conduct, and ensuring that the review meets its objectives.
  - The **Recorder** documents anomalies, action items, decisions, and recommendations made by the review team.
  - **Technical staff** are active participants in the review and evaluation of the software product.
  - **Management staff** may participate for the purpose of identifying issues that require management resolution.
  - **Customer or user representatives** may fill roles determined by the Review Leader prior to the review.
  - A single participant may fill more than one role, as appropriate.

- The objectives of an FTR are:
  1. to uncover errors in function, logic, or implementation for any representation of the software;
  2. to verify that the software under review meets its requirements;
  3. to ensure that the software has been represented according to predefined standards;
  4. to achieve software that is developed in a uniform manner; and
  5. to make projects more manageable.
- In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation.
- The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.
- The FTR is actually a class of reviews that includes walkthroughs and inspections.
- Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended.

Review Meeting :

- Every review meeting should stand by the following constraints:
  - Between three and five people (typically) should be involved in the review.
  - Advance preparation should occur but should require no more than two hours of work for each Person.
  - The duration of the review meeting should be less than two hours.
- Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software.
- For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components.
- By narrowing the focus, the FTR has a higher likelihood of uncovering errors.

- The focus of the FTR is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component).
- The individual who has developed the work pro duct-the producer- informs the project leader that the work product is complete and that a review is required.
- The project leader contacts a review leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.
- Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

- The review meeting is attended by the review leader, ail reviewers, and the producer.
- One of the reviewers takes on the role of a recorder, that is, the individual who records (in writing) all important issues raised during the review.
- The FTR begins with an introduction of the agenda and a brief introduction by the producer.
- The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation.
- when valid problems or errors are discovered, the recorder notes each.

- At the end of the review, all attendees of the FTR must decide whether to:
  1. accept the product without further modification ,
  2. reject the product due to severe errors (once corrected, another review must be performed), or
  3. accept the product provisional$ (minor errors have been encountered and must be corrected, but no additional review will be required).
- After the decision is made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting & Record keeping

- During the FTR, a reviewer (the recorder) actively records all issues that have been raised.
- These are summarized at the end of the review meeting, and, a review issues list is produced.
- In addition, a formal technical review summary report is completed.
- A review summary report answers three questions:
  1. What was reviewed?
  2. Who reviewed it?
  3. What were the findings and conclusions?
- The review summary report is a single page form (with possible attachments).
- It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

- The review issues list serves two purposes:
  1. to identify problem areas within the product and
  2. to serve as an action item checklist that guides the producer as corrections are made.
- An issues list is normally attached to the summary report.
- You should establish a follow-up procedure to ensure that items on the issues list have been properly corrected.
- Unless this is done, it is possible that issues raised can "fall between the cracks."
- One approach is to assign the responsibility for follow-up to the review leader

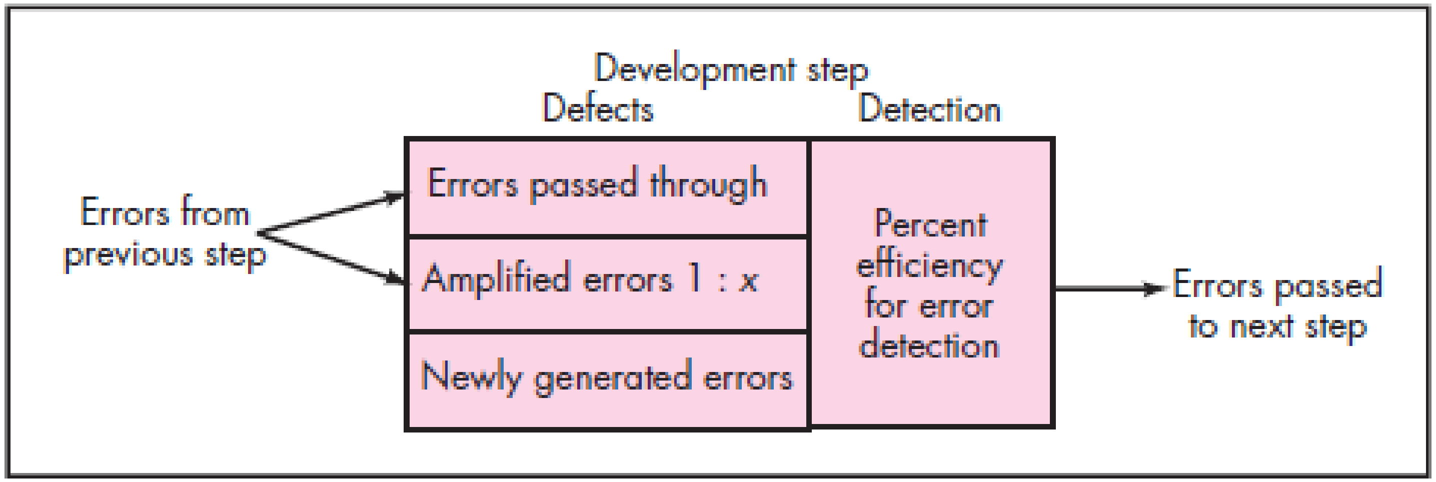
COST IMPACT OF SOFTWARE DEFECTS

- Within the context of the software process, the terms *defect* and *fault* are synonymous. Both imply a quality problem that is discovered *after* the software has been released to end users.
- The term *error* to depict a quality problem that is discovered by software engineers (or others) *before* the software is released to the end user.
- The primary objective of technical reviews is to find errors during the process so that they do not become defects after release of the software.
- The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

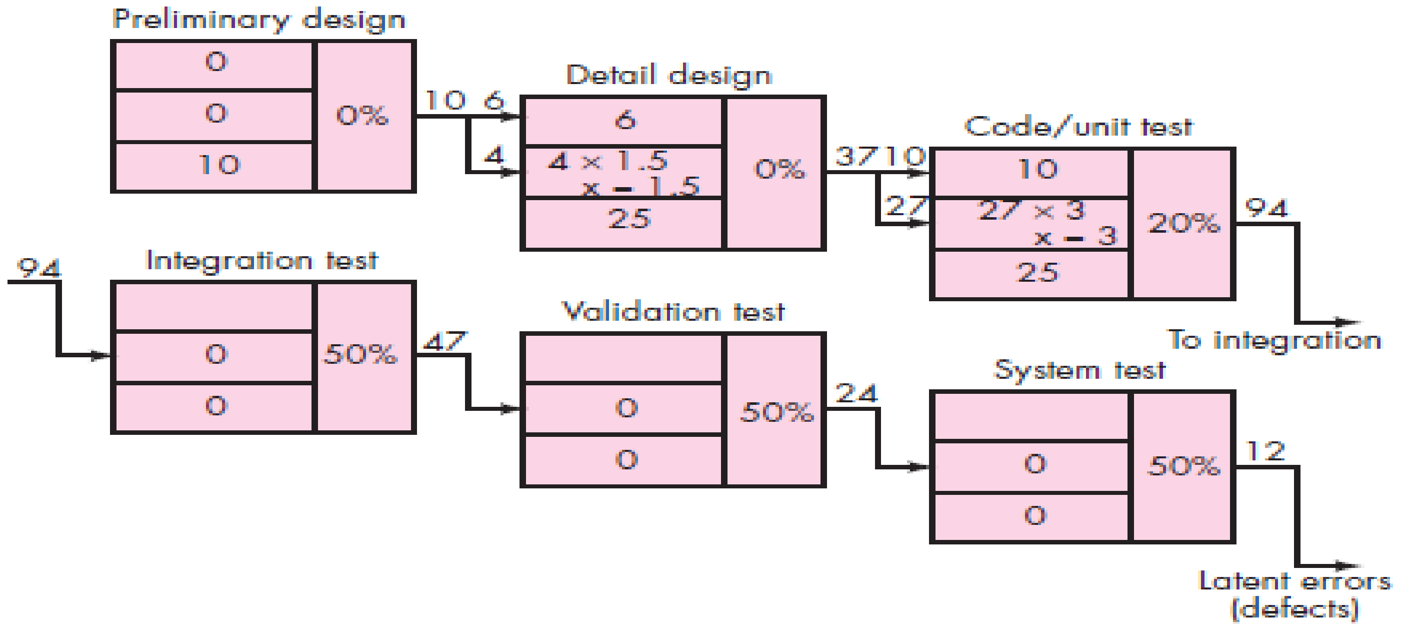
- A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process.
- However, review techniques have been shown to be up to 75 percent effective in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent activities in the software process.

DEFECT AMPLIFICATION AND REMOVAL

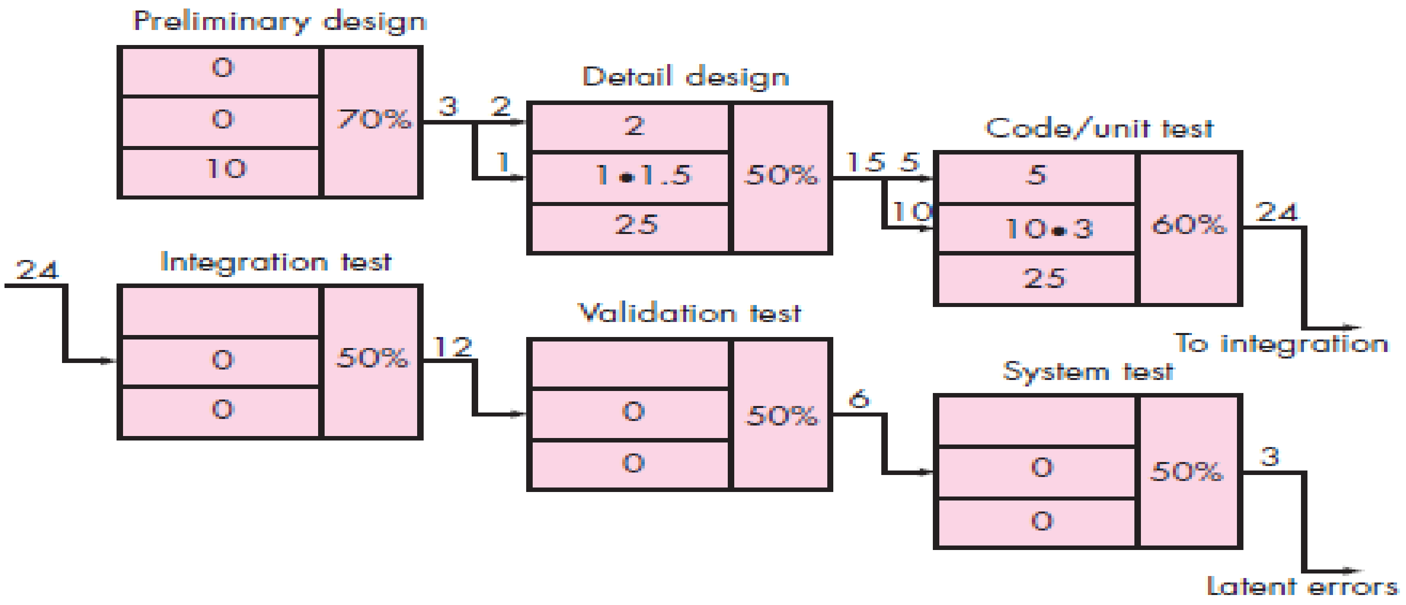
- A *defect amplification model* can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process.



- A box represents a software engineering action. During the action, errors may be inadvertently generated.
- Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work.
- The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.
- To conduct reviews, you must expend time and effort, and your development organization must spend money



Preliminary design
0
0
10
0%
10
6
4
Detail design
6
4 × 1.5
x = 1.5
25
0%
37
10
27
Code/unit test
10
27 × 3
x = 3
25
20%
94
To integration
94
Integration test
0
0
50%
47
Validation test
0
0
50%
24
System test
0
0
50%
12
Latent errors
(defects)



Preliminary design
0
0
10
70%
3
2
1
Detail design
2
1•1.5
25
50%
15
5
10
Code/unit test
5
10•3
25
60%
24
To integration
24
Integration test
0
0
50%
12
Validation test
0
0
50%
6
System test
0
0
50%
3
Latent errors

REVIEW METRICS AND THEIR USE

- The following review metrics can be collected for each review that is conducted:
  - *Preparation effort, Ep*—the effort (in person-hours) required to review a work product prior to the actual review meeting
  - *Assessment effort, Ea*—the effort (in person-hours) that is expended during the actual review
  - *Rework effort, Er*—the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
  - *Work product size,WPS*—a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)
  - *Minor errors found, Errminor*—the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
  - *Major errors found, Errmajor*—the number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)

Introduction to Software Configuration Management

- Change is inevitable when computer software is built. And change increases the level of confusion when you and other members of a software team are working on a project.
- Confusion arises when changes are not analyzed before they are made, recorded before they are implemented, reported to those with a need to know, or controlled in a manner that will improve quality and reduce error.
- The art of coordinating software development to minimize . . . confusion is called configuration management. Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

- Software configuration management (SCM) is an umbrella activity that is applied throughout the software process.
- Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest.

Why SCM?

- The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms), (2) work products that describe the computer programs (targeted at various stakeholders), and (3) data or content
- As software engineering work progresses, a hierarchy of *software configuration items (SCIs)—a named element of information that can be as small as a single UML* diagram or as large as the complete design document—is created.
- Change may occur at any time, for any reason. In fact, the
- *First Law of System Engineering [Ber80] states:"No matter where you are in the system* life cycle, the system will change, and the desire to change it will persist throughout the life cycle."
- What is the origin of these changes?

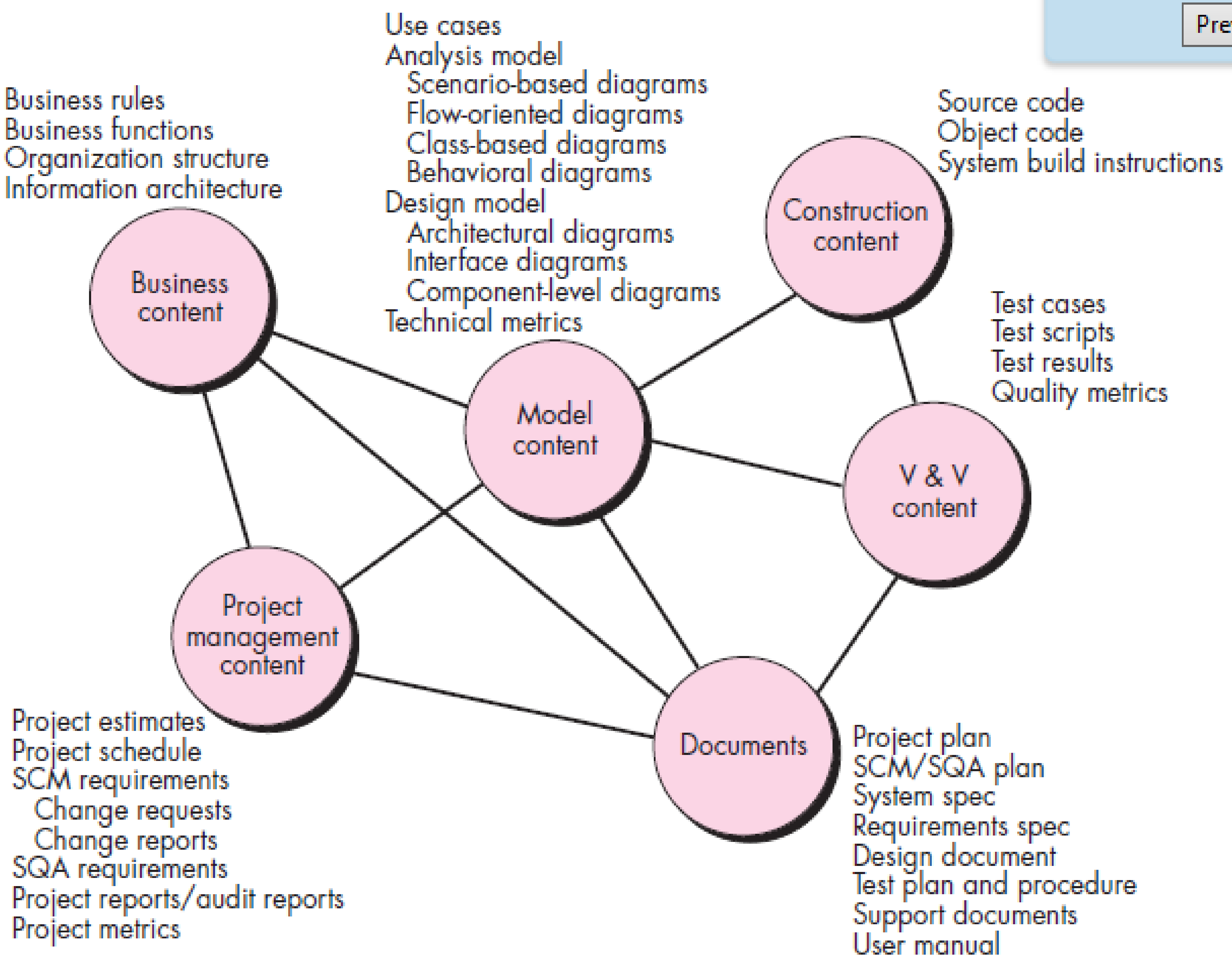
- New business or market conditions dictate changes in product requirements or business rules.
  - New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
  - Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
  - Budgetary or scheduling constraints cause a redefinition of the system or product.
- Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process.

THE SCM REPOSITORY

- In the early days of software engineering, software configuration items were maintained as paper documents (or punched computer cards!), placed in file folders or three-ring binders, and stored in metal cabinets. This approach was problematic for many reasons: (1) finding a configuration item when it was needed was often difficult, (2) determining which items were changed, when and by whom was often challenging, (3) constructing a new version of an existing program was time consuming and error prone, and (4) describing detailed or complex relationships between configuration items was virtually impossible.
- Today, SCIs are maintained in a project database or repository.

- The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner.
- It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration. In addition, the SCM repository provides a hub for the integration of software tools, is central to the flow of the software process, and can enforce uniform structure and format for software engineering work products.
- The repository is defined in terms of a meta-model. The *meta-model determines how information is stored in the repository, how data* can be accessed by tools and viewed by software engineers

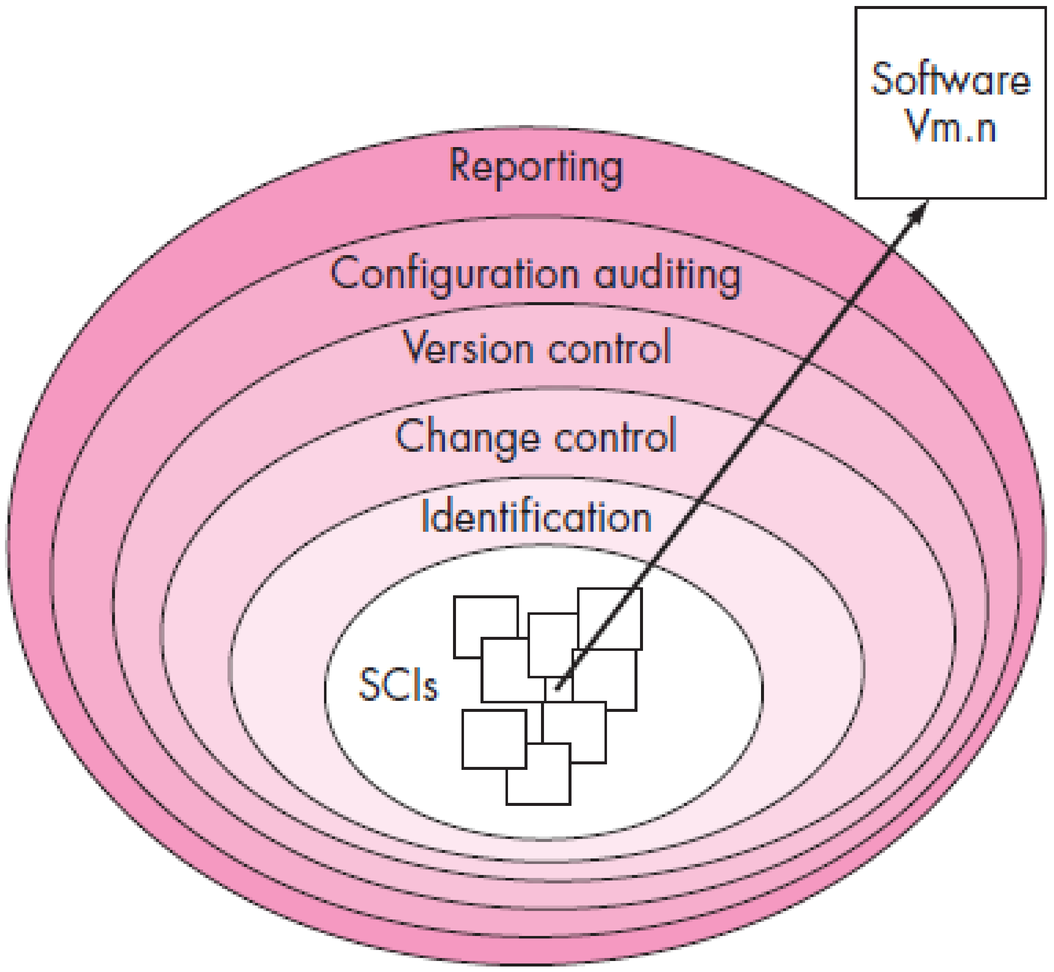
- **General Features and Content:**
  - Versioning.
  - Dependency tracking and change management
  - Requirements tracing
  - Configuration management
  - Audit trails


Use cases
Analysis model
Scenario-based diagrams
Flow-oriented diagrams
Class-based diagrams
Behavioral diagrams
Design model
Architectural diagrams
Interface diagrams
Component-level diagrams
Technical metrics
Business rules
Business functions
Organization structure
Information architecture
Source code
Object code
System build instructions
Construction content
Business content
Test cases
Test scripts
Test results
Quality metrics
Model content
V & V content
Project management content
Documents
Project estimates
Project schedule
SCM requirements
Change requests
Change reports
SQA requirements
Project reports/audit reports
Project metrics
Project plan
SCM/SQA plan
System spec
Requirements spec
Design document
Test plan and procedure
Support documents
User manual

Prev

THE SCM PROCESS

- The software configuration management process defines a series of tasks that have four primary objectives: (1) to identify all items that collectively define the software configuration, (2) to manage changes to one or more of these items, (3) to facilitate the construction of different versions of an application, and (4) to ensure that software quality is maintained as the configuration evolves over time.
- Referring to the figure, SCM tasks can viewed as concentric layers. SCIs flow outward through these layers throughout their useful life, ultimately becoming part of the software configuration of one or more versions of an application or system.
- As an SCI moves through a layer, the actions implied by each SCM task may or may not be applicable


Software
Vm.n
Reporting
Configuration auditing
Version control
Change control
Identification
SCIs

- For example, when a new SCI is created, it must be identified.
- However, if no changes are requested for the SCI, the change control layer does not apply. The SCI is assigned to a specific version of the software (version control mechanisms come into play).
- A record of the SCI (its name, creation date, version designation, etc.) is maintained for configuration auditing purposes and reported to those with a need to know.