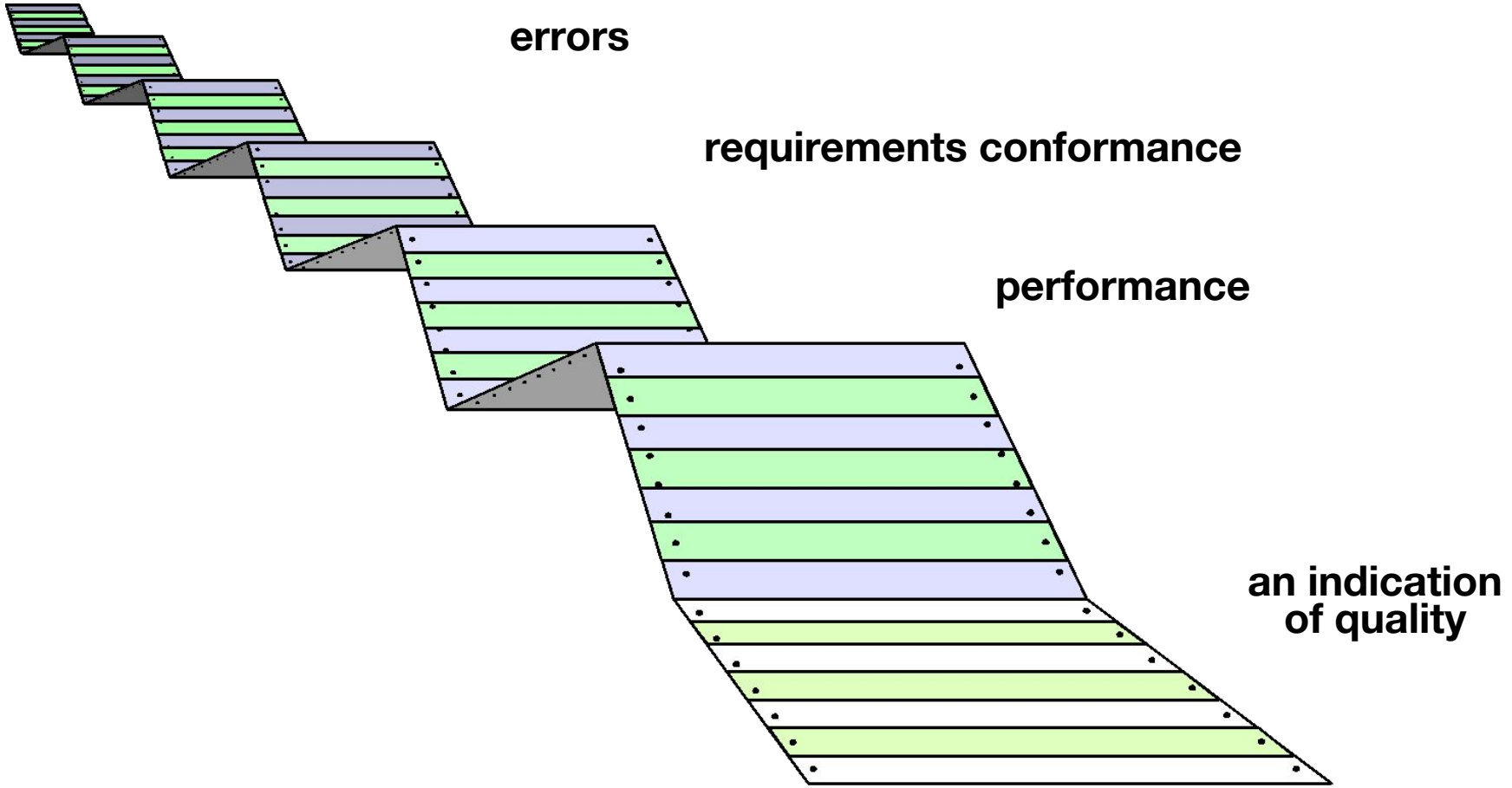
UNIT V

Software Testing

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

What Testing Shows



Strategic Approach

- 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.

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 [Boe81] states this another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

Who Tests the Software?

developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

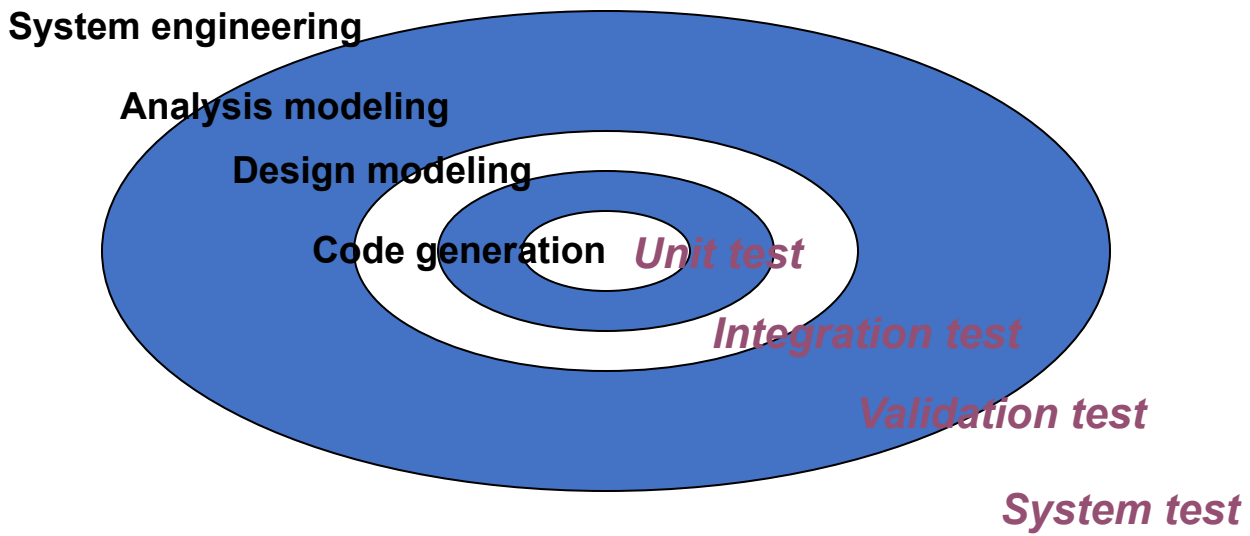
independent tester

Must learn about the system

but, will attempt to break it

and, is driven by quality

Testing Strategy



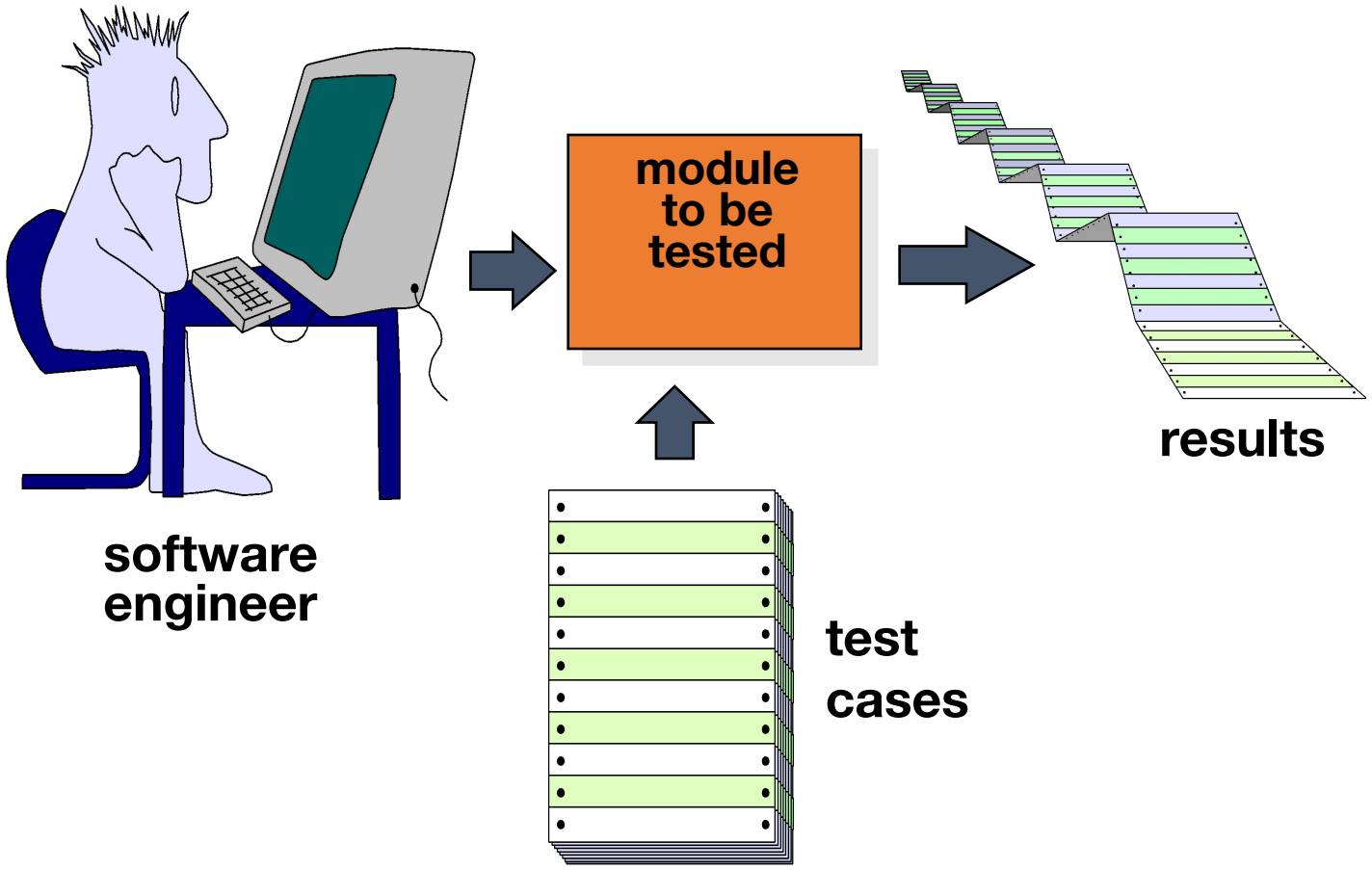
Testing Strategy

- We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

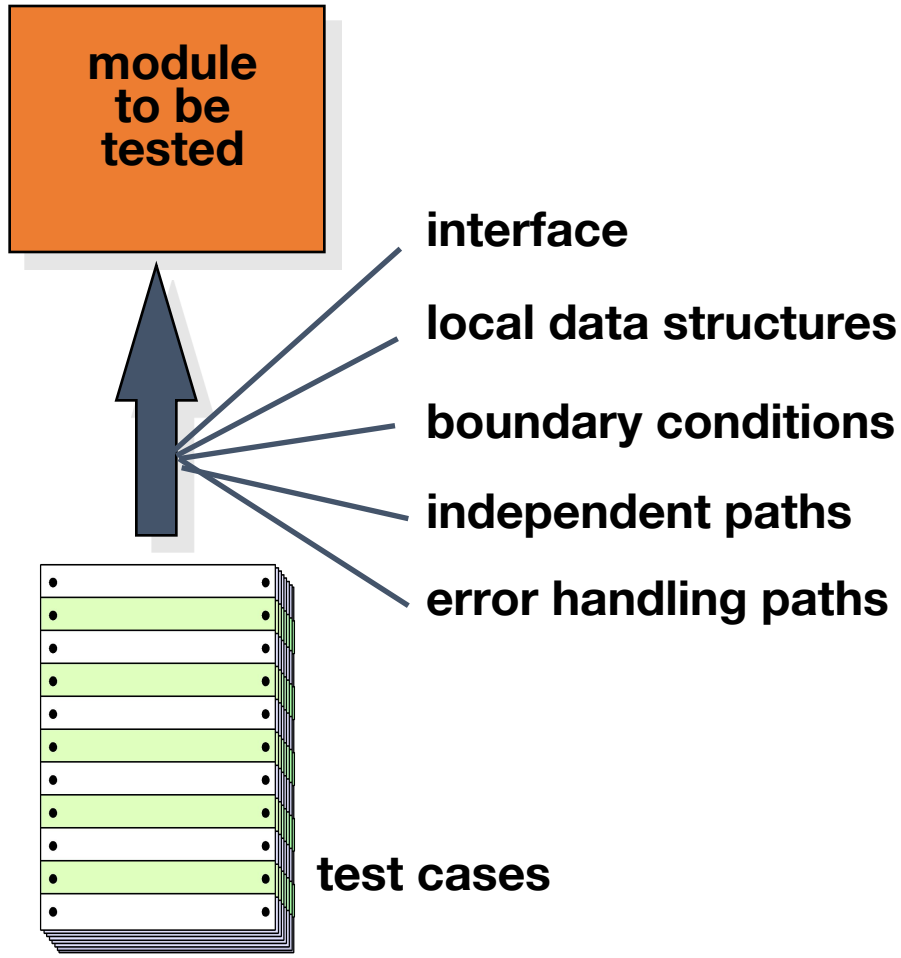
Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective technical reviews as a filter prior to testing
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

Unit Testing



Unit Testing



Contd..

Unit Testing

- The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

Unit-test considerations:-

1. The module interface is tested to ensure proper information flows (into and out).
2. Local data structures are examined to ensure temporary data store during execution.
3. All independent paths are exercised to ensure that all statements in a module have been executed at least once.
4. Boundary conditions are tested to ensure that the module operates properly at boundaries. Software often fails at its boundaries. 5. All error-handling paths are tested.

If data do not enter and exit properly, all other tests are controversial. Among the potential errors that should be tested when

error handling is evaluated are:

- Error description is unintelligible,
- Error noted does not correspond to error encountered,
- Error condition causes system intervention prior to error handling,
- exception-condition processing is incorrect,
- Error description does not provide enough information to assist in the location of the cause of the error

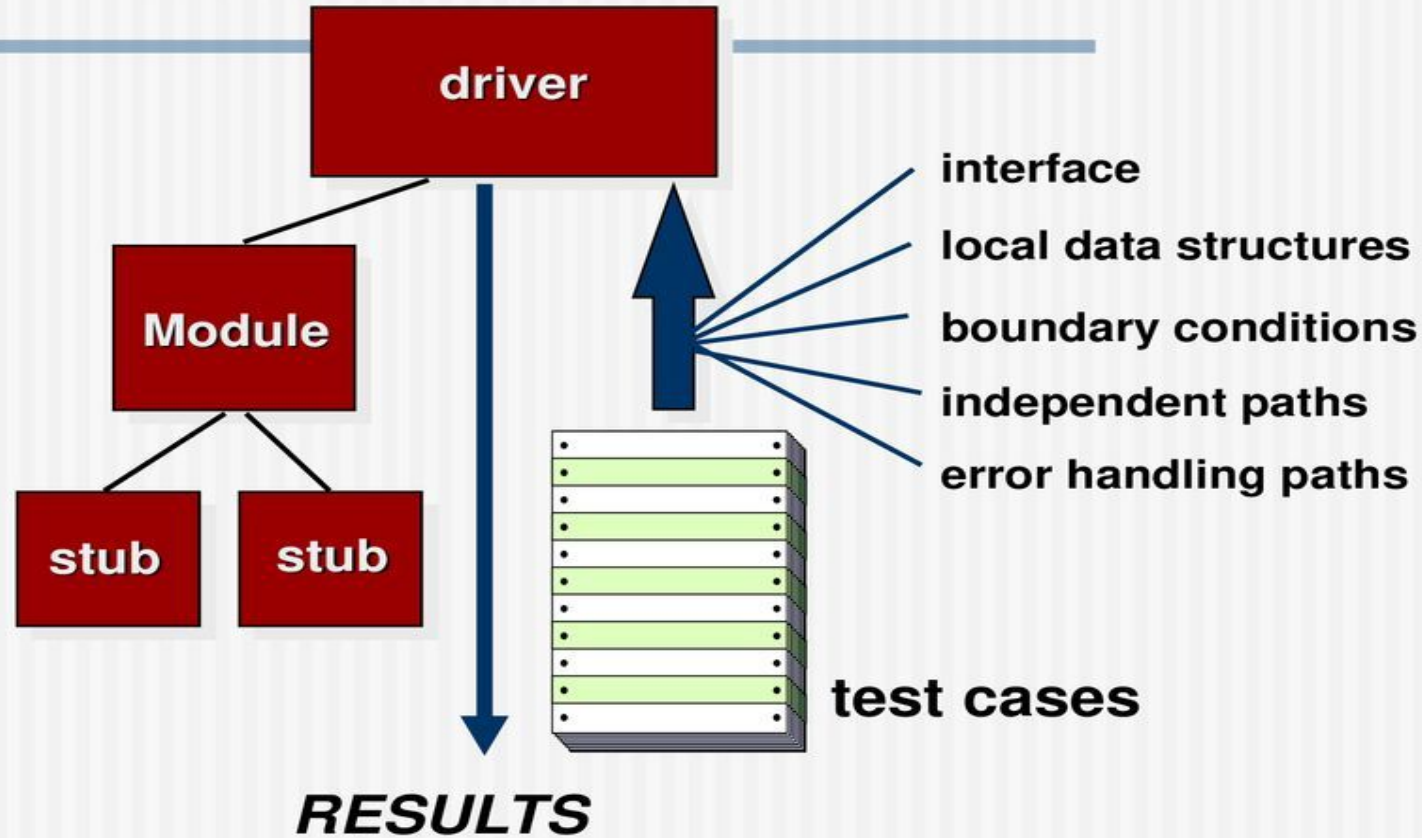
Unit-test procedures

The design of unit tests can occur before coding begins or after source code has been generate. Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.

✔ Driver is nothing more than a “main program” that accepts test case data, passes such data to the component (to be tested), and prints relevant results.

✔ Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing

Unit Test Environment

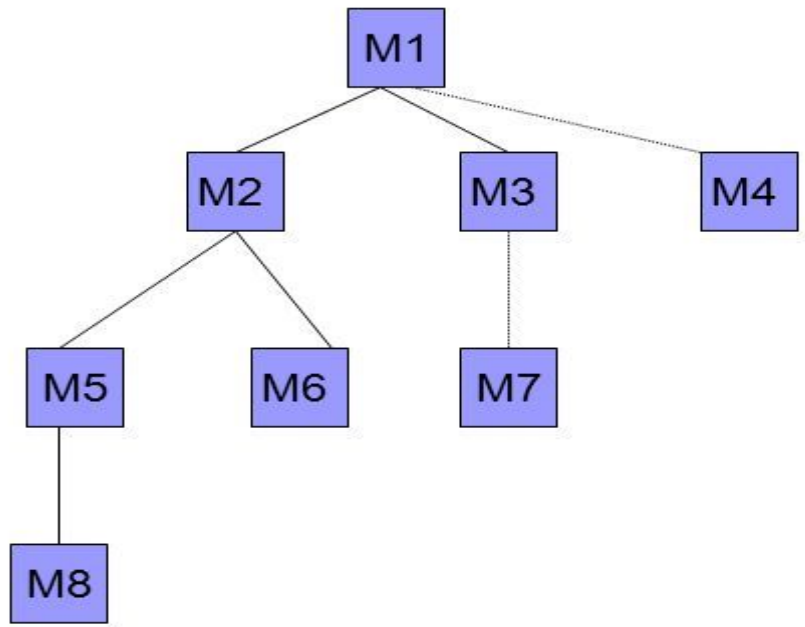


Integration Testing

- Data can be lost across an interface; one component can have an inadvertent, adverse effect on another; sub functions, when combined, may not produce the desired major function.
- The objective of Integration testing is to take unit-tested components and build a program structure that has been dictated by design.
- The program is constructed and tested in small increments, where errors are easier to isolate and correct

Top-Down Integration Testing

- Modules are integrated by moving downward through control hierarchy.
- Modules subordinate to main control module are incorporated
  - Depth-first
  - Breadth-first



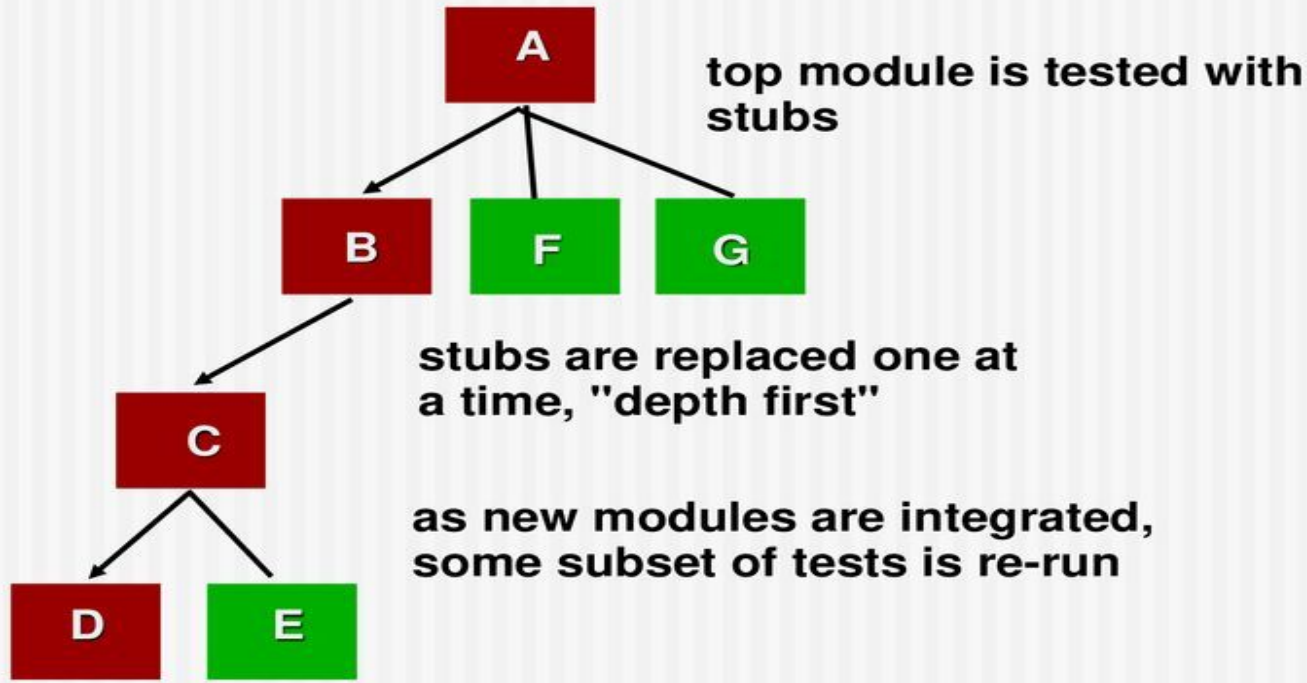
Pairs: What get's tested first in breadth first?

Top-down integration testing

- It is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy. Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced

Top Down Integration



Bottom-up integration

- Begins construction and testing with components at the lowest levels in the program structure.
- Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated

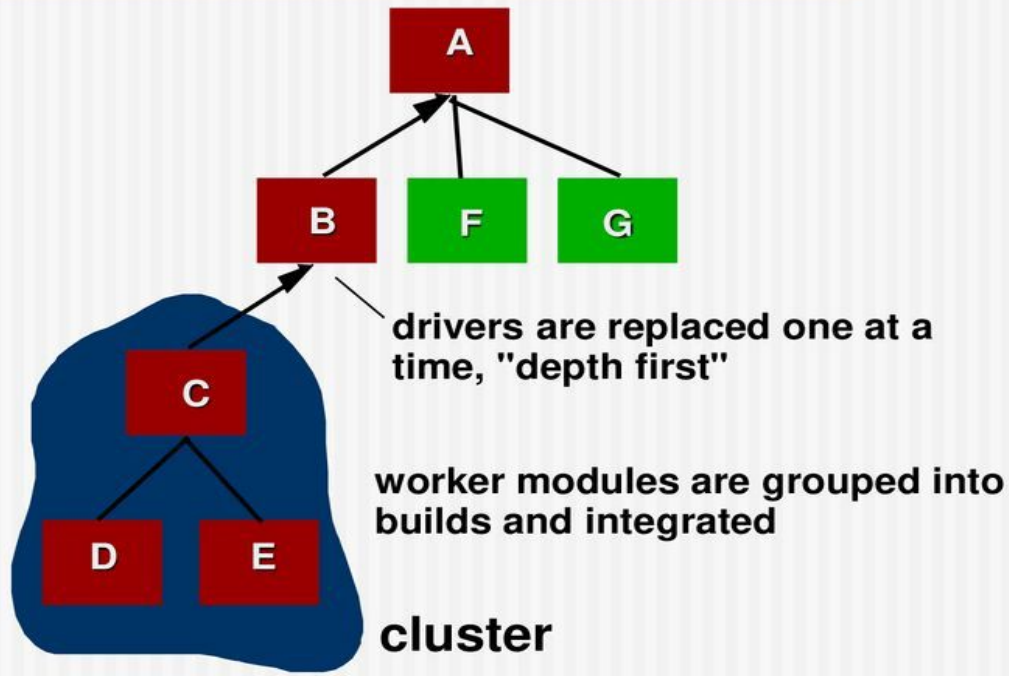
1.Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub function.

2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

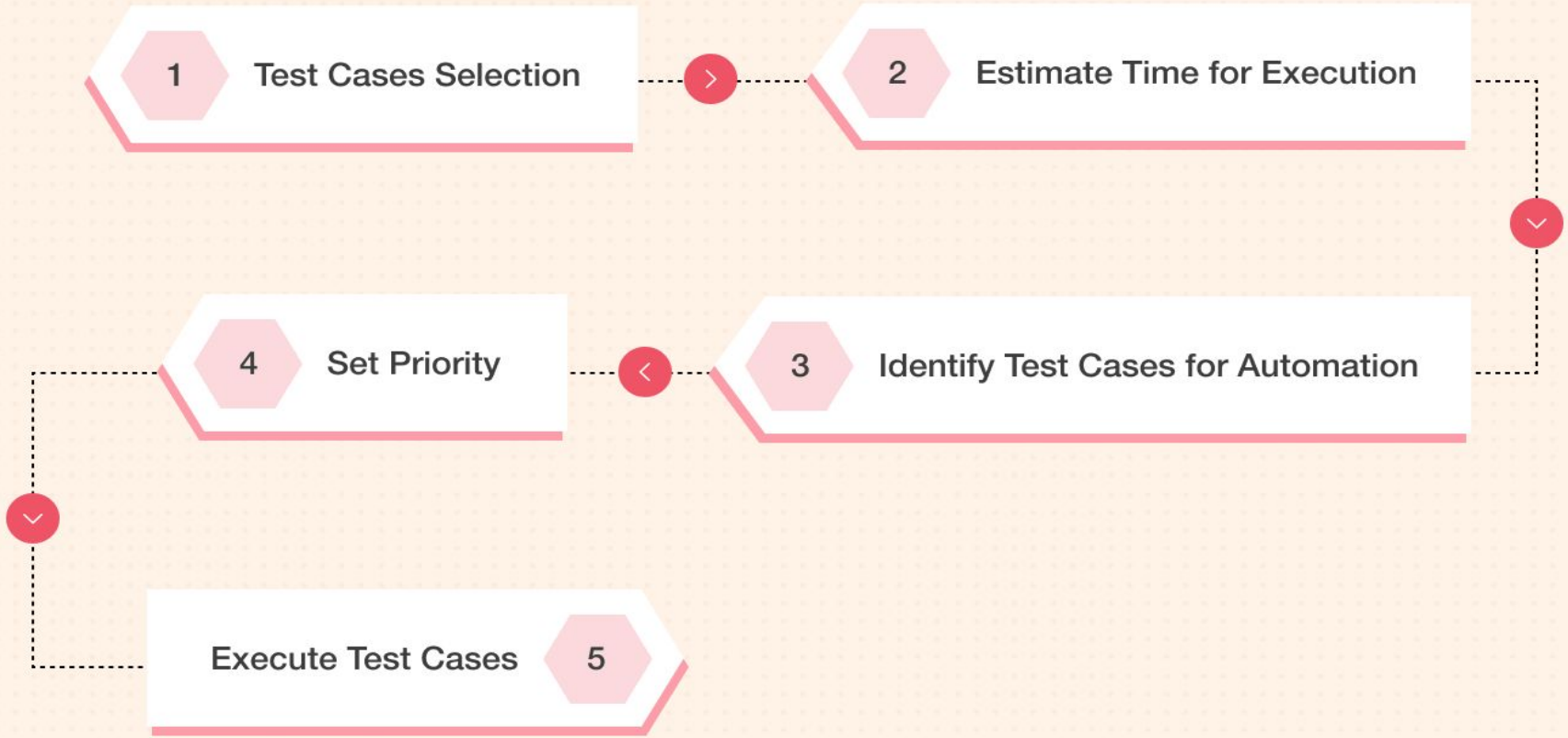
Bottom-Up Integration



Regression Testing

- *Regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

How to do Regression Testing?



SIMFORM

Smoke Testing

- Smoke Testing comes into the picture at the time of receiving build software from the development team.
- The purpose of smoke testing is to determine whether the build software is testable or not. It is done at the time of "building software.
- The focus of Smoke Testing is on the workflow of the core and primary functions of the application.
- In the smoke testing, we only focus on the positive flow of the application and enter only valid data, not the invalid data.
- Smoke testing does not require to design test cases. There's need only to pick the required test cases from already designed test cases.

Smoke Testing

- A common approach for creating "daily builds" for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a "build."
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

Benefits of smoke testing

- Integration risk is minimized
- The quality of the end product is improved
- Error diagnosis and correction are simplified
- Progress is easier to asses

OO Testing Strategy

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

Validation Testing

- The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.
- Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

Verification	Validation
Are we implementing the system right?	Are we implementing the right system?
Evaluating products of a development phase	Evaluating products at the closing of the development process
The objective is making sure the product is as per the requirements and design specifications	The objective is making sure that the product meets user's requirements
Activities included: reviews, meetings, and inspections	Activities included: black box testing, white box testing, and grey box testing
Verifies that outputs are according to inputs or not	Validates that the users accept the software or not
Items evaluated: plans, requirement specifications, design specifications, code, and test cases	Items evaluated: actual product or software under test
Manual checking of the documents and files	Checking the developed products using the documents and files

Alpha vs. Beta Testing

Alpha Testing

- Is conducted at the Developer's site by a customer
- The developer would supervise
- Is conducted in a controlled environment

Beta Testing

- Is conducted at customer's site by the end user of the software
- The developer is generally not present
- Is conducted in a "live" environment

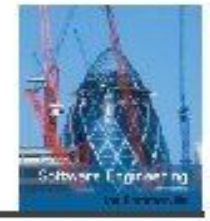
SYSTEM TESTING

- *System testing is actually a series of different tests whose primary purpose is to* fully exercise the computer-based system.
- It verifies that system elements have been properly integrated and perform allocated functions.

System Testing

- **Recovery testing:** is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- **Security testing:** attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- **Stress testing:** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume (e.g. special tests may be designed that generate ten interrupts per second, when one or two is the average rate).

3. Performance testing

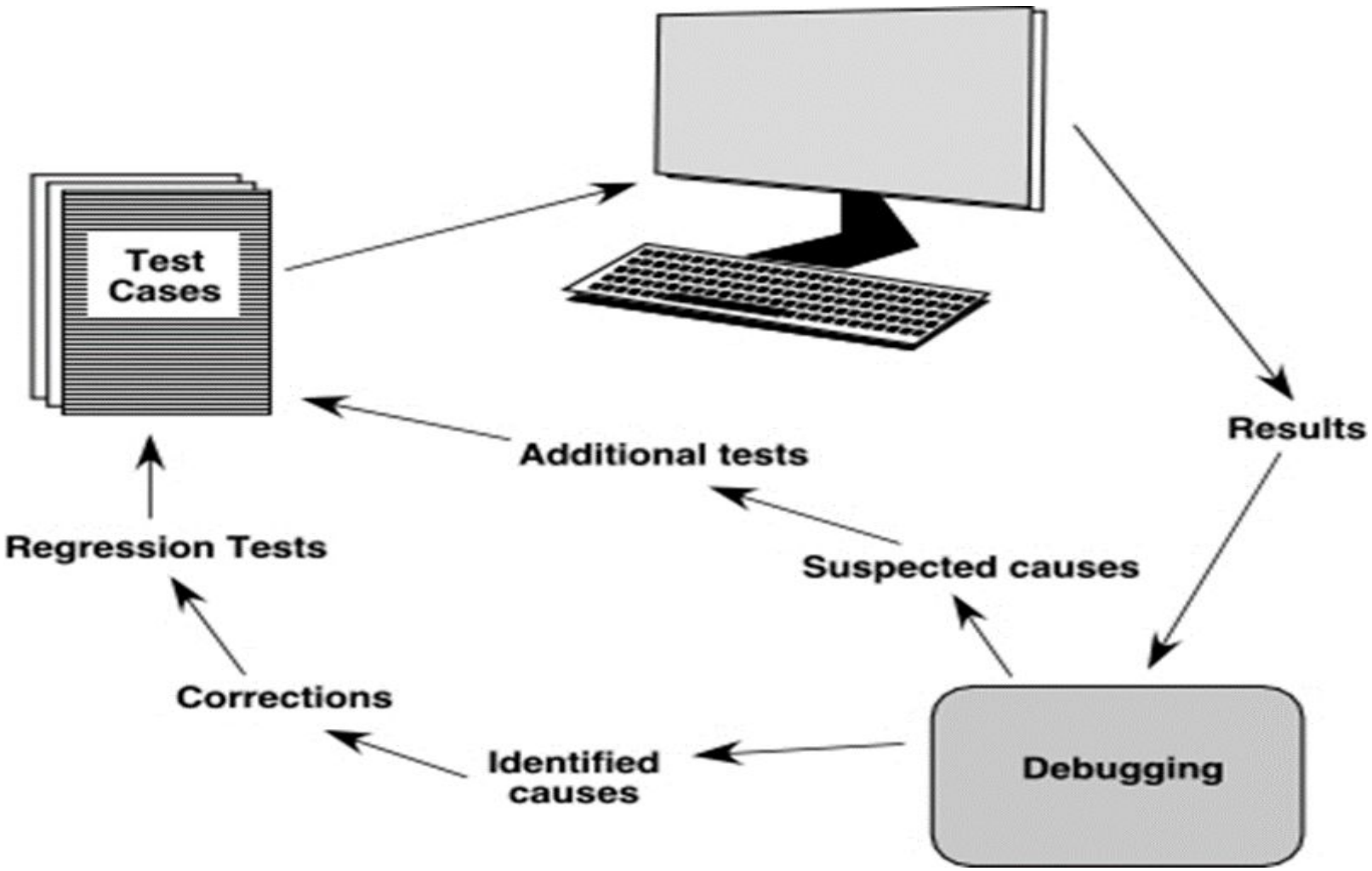


- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour. (Covered in Chapter 11)

The Art of Debugging

- Debugging occurs as a consequence of successful testing.
- That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art.
- A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem.

The Debugging Process

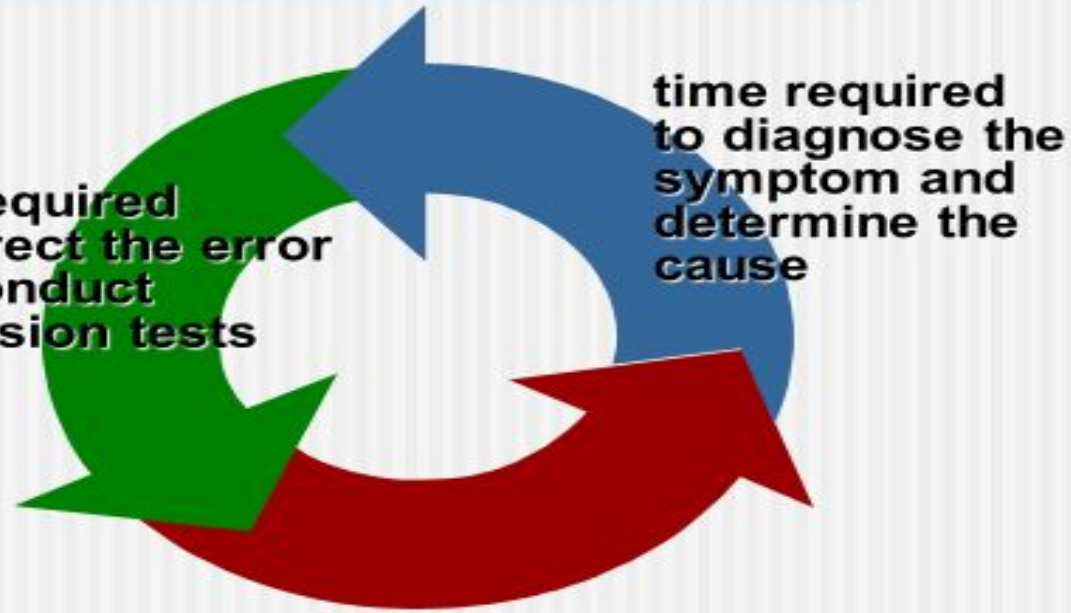


Contd..

The debugging process will always 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.

Debugging Effort



Symptoms & Causes

1.The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.

2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).

4. The symptom may be caused by human error that is not easily traced.

5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Psychological Considerations

- Debugging is one of the more frustrating parts of programming.
- It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake.
- Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty.

Debugging Techniques

Brute force

- ✔ The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error.
- ✔ We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements.

Backtracking

- ✔ 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 site of the cause is found.
- ✔ Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

Cause Elimination

- uses induction or deduction:
  - Data related to the error occurrence are organised to isolate potential causes
  - A cause hypothesis is devised and the above data are used to prove or disprove the hypothesis
- Alternatively:
  - a list of all the possible causes is developed and tests are conducted to eliminate each

Correcting the Error

- *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.
- *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.
- *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.

Testability

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

What is a "Good" Test?

- 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	White Box Testing
1	Black box testing is the Software testing method which is used to test the software without knowing the internal structure of code or program.	White box testing is the software testing method in which internal structure is being known to tester who is going to test the software.
2	This type of testing is carried out by testers.	Generally, this type of testing is carried out by software developers.
3	Implementation Knowledge is not required to carry out Black Box Testing.	Implementation Knowledge is required to carry out White Box Testing.
4	Programming Knowledge is not required to carry out Black Box Testing.	Programming Knowledge is required to carry out White Box Testing.
5	Testing is applicable on higher levels of testing like System Testing, Acceptance testing.	Testing is applicable on lower level of testing like Unit Testing, Integration testing.
6	Black box testing means functional test or external testing.	White box testing means structural test or interior testing.

White Box Testing

- Knowing the internal workings of a product, tests are performed to check the workings of all possible logic paths
- White box testing uses the control structure described as part of the component level design to derive test cases.
- Using white box testing methods, the software engineer can derive test cases that:
  - Guarantee that all independent paths within a module have been exercised at least once.
  - Exercise all logical decisions on their true and false sides.
  - Execute all loops at their boundaries and within their operational bounds.
  - Exercise internal data structures to ensure their validity.

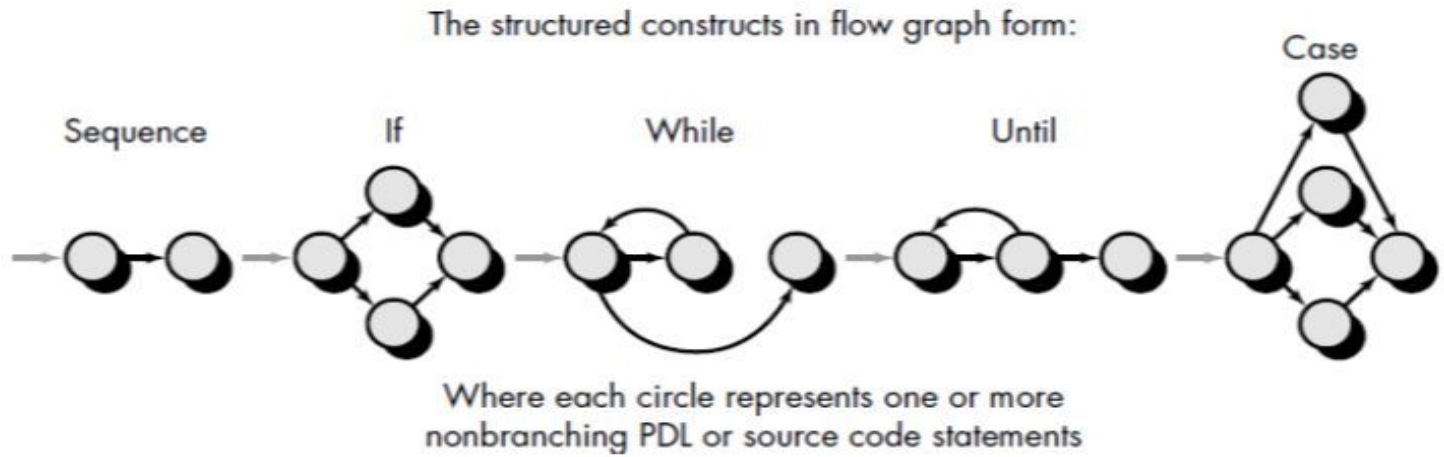
Path Testing

- Path testing is a Structural Testing method that involves using the source code of a program to attempt to find every possible executable path.
- The idea is that are able to test each individual path from source code is as many way as possible in order to maximize the coverage of each test case.
- Therefore, we use knowledge of the source code to define the test cases and to examine outputs.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing

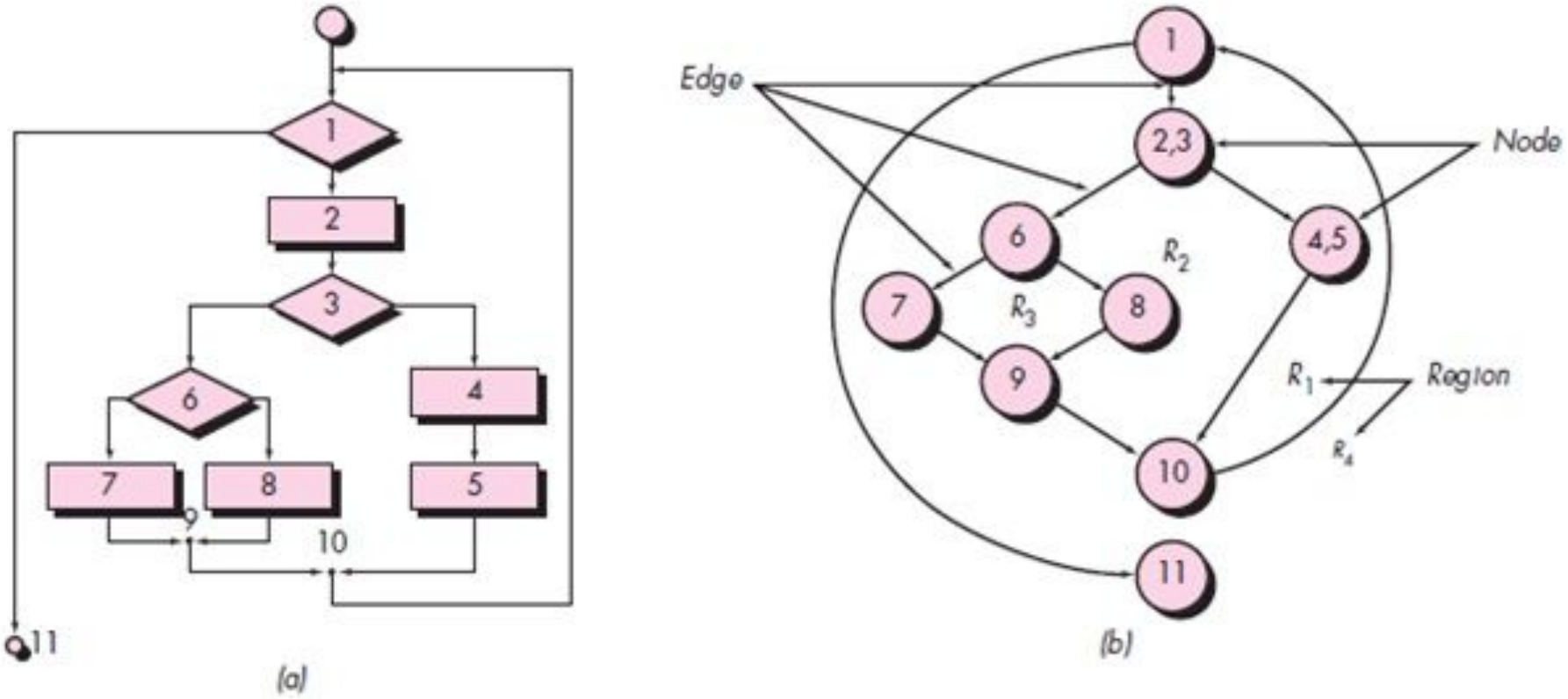
Path Testing

- ***Program graph*** is a **directed graph** in which
  - **nodes** are either **entire statements or fragments of a statement**,
  - **edges** represent **flow of control.**
- The importance of the program graph is that **program executions correspond to paths from the source to the sink nodes.**

FIGURE 17.1
Flow graph notation



Flow chart & Flowgraph



Gambar 2. Flowchart dan flow graph

Independent program path

- Any path through the program that introduces at least one new set of processing statements or a new condition.
- An independent path must move along at least one edge that has not been traversed before the path is defined.

- The basic set is not unique.
- How do we know how many paths to look for?
  - Cyclomatic complexity

Cyclomatic Complexity

- A testing metric used for measuring the complexity of a software program. It is a quantitative measure of independent paths in the source code of a software program. Cyclomatic complexity can be calculated by using control flow graphs or with respect to functions, modules, methods or classes within a software program.

Use the following formula to calculate cyclomatic complexity (CYC):

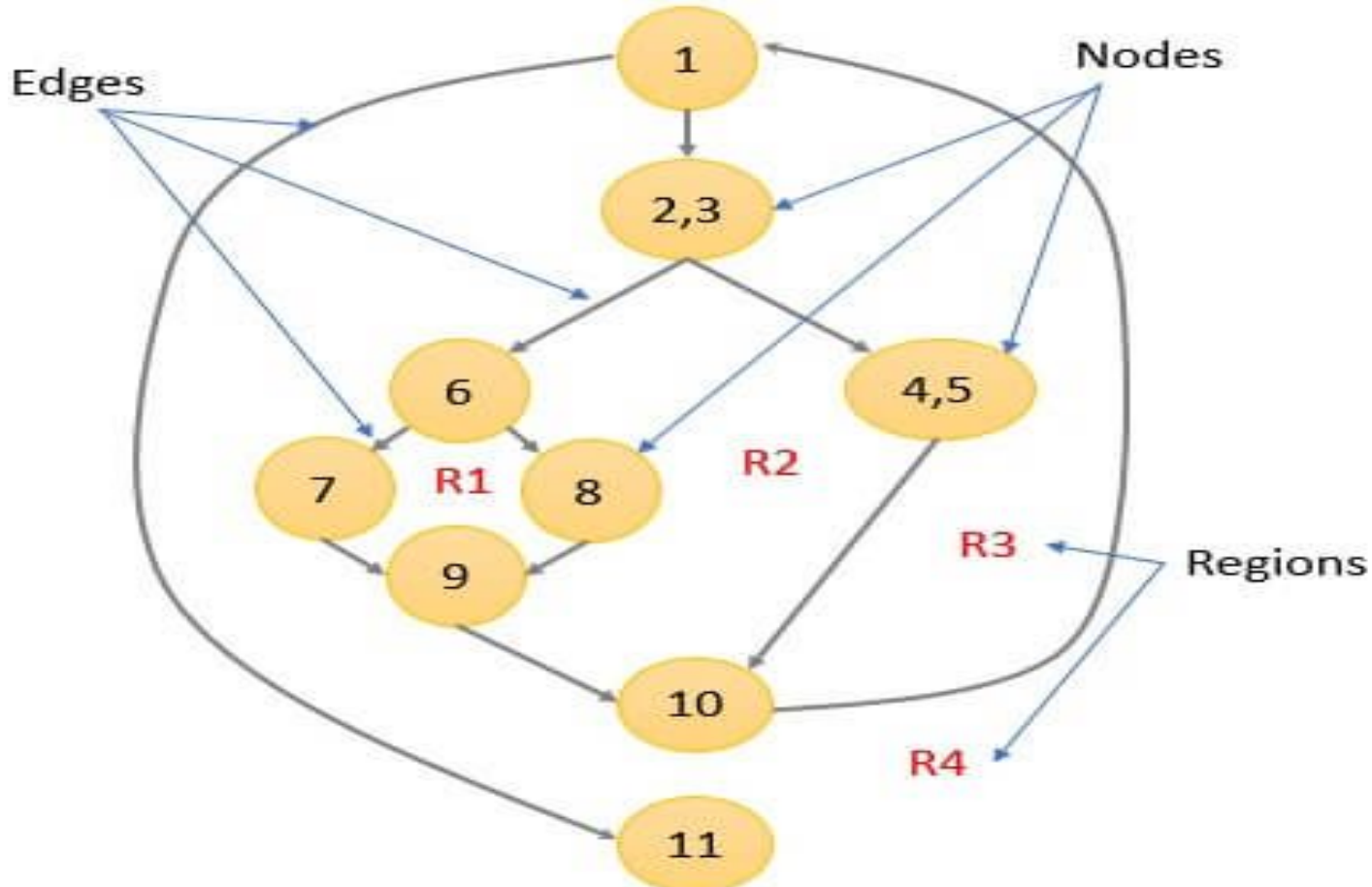
$$CYC = E - N + 2P$$

P = Number of disconnected parts of the flow graph (e.g. a calling program and a subroutine

E = Number of edges (transfers of control)

N = Number of nodes (sequential group of statements containing only one transfer of control)

This translates to the number of decisions + 1.



Flow Graph for above Flow Chart

- The cyclomatic complexity V(G) can be calculated with the formulas below:

- V(G) = E – N + 2 // E is no. of edges in the flow graph, N is no. of nodes in the flow graph.
- V(G) = 11 edges – 9 nodes + 2 = 4
- V(G) = P + 1 // P is no. of predicate node
- V(G) = 3 predicate nodes + 1 = 4
- V(G)= no. of regions of a flow graph =4

Control Structure Testing

- White-box testing technique focusing on control structures present in the software.
- The basis path testing technique is one of a number of techniques for control structure testing. Basis path testing is simple and effective; however, it is not sufficient in itself.
- Following are some variations on control structure testing; they broaden testing coverage and improve quality of white-box testing –
  - Condition testing
  - Data flow testing
  - Loop testing

Control Structure Testing

1. **Condition Testing**

   a test case design method that exercise the logical conditions contained in a program module

2. **Data Flow Testing**

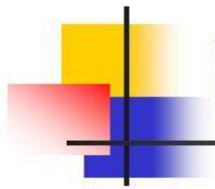
   a method to select a test path of a program according to the location of definitions and uses of variable in the program

3. **Loop Testing**

   a white box testing technique that focuses exclusively on the validity of loop construct

Control Structure Testing

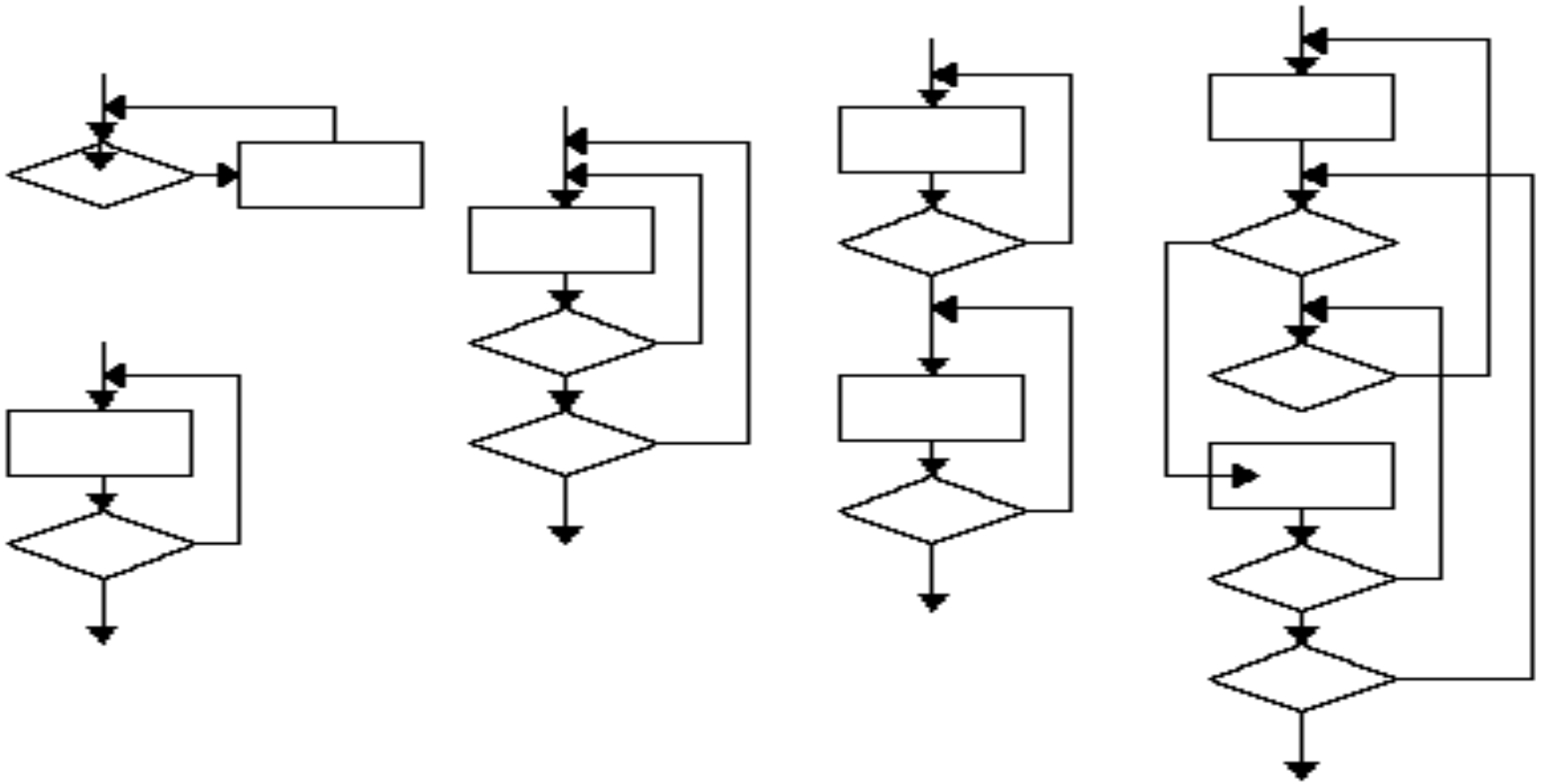
- **Condition testing:**
- Condition Testing is a test case design method that exercises the logical conditions contained in a program module.
- Focuses on testing each condition in the program to ensure that it does not contain errors.
- Types of errors found include:
  - Boolean operator error (OR, AND, NOT)
  - Boolean variable error
  - Boolean parenthesis error
  - Relational operator error (>,<,=,!=,...)
  - Arithmetic expression error



Data Flow Testing

- Testing All-Nodes and All-Edges in a control flow graph may miss significant test cases
- Testing All-Paths in a control flow graph is often too time-consuming
- Can we select a subset of these paths that will reveal the most faults?
- Data Flow Testing focuses on the points at which variables receive values and the points at which these values are used

- **Loops** are the cornerstone for the vast majority of all algorithms implemented in software.
- ***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
Nested
Concatendate
Unstructured

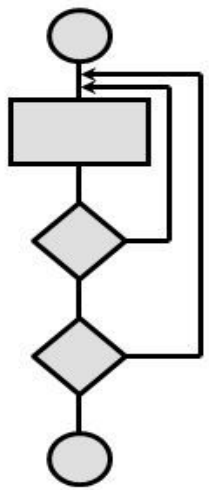
Loop Testing: Simple Loops

Minimum conditions—Simple Loops

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

where n is the maximum number of allowable passes

Nested loops



- Extend simple loop testing
- Reduce the number of tests:
  - start at the innermost loop; set all other loops to minimum values
  - conduct simple loop test; add out of range or excluded values
  - work outwards while keeping inner nested loops to ***typical*** values
  - continue until all loops have been tested

Black box testing

- Black box testing involves testing a system with no prior knowledge of its internal workings.
- A tester provides an input, and observes the output generated by the system under test.
- This makes it possible to identify how the system responds to expected and unexpected user actions, its response time, usability issues and reliability issues.

Black box testing

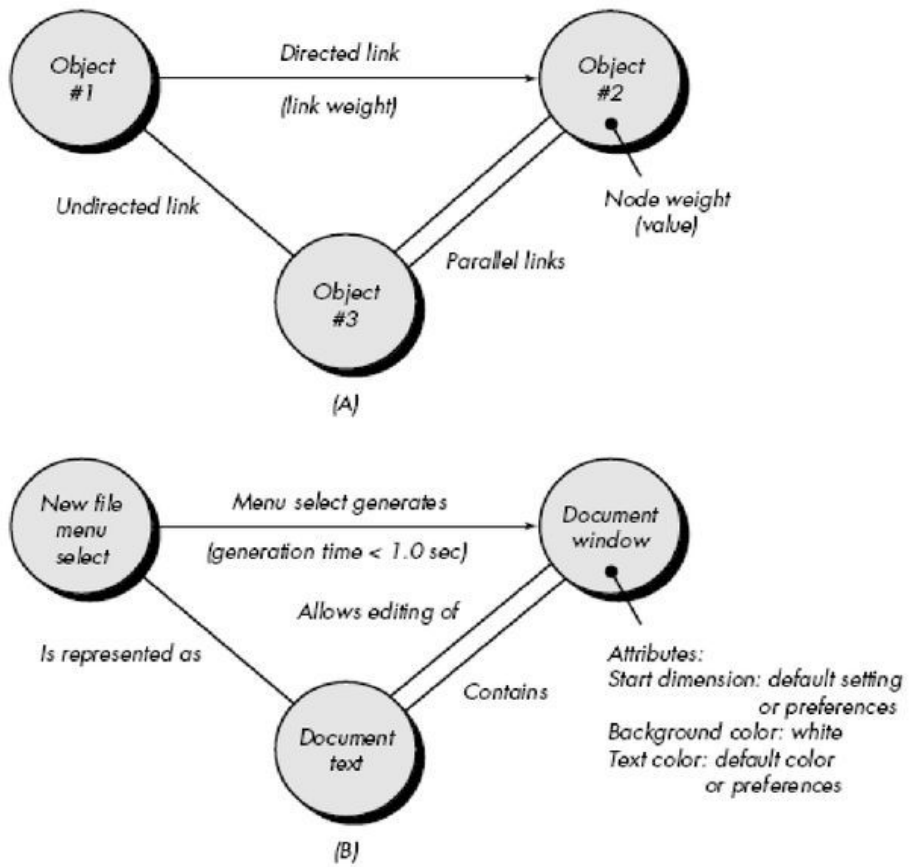
Incorrect or missing functions,interface errors,errors in data structures,performance errors and termination errors

Graph Based Testing Method

- Software testing begins by creating graph of important objects and their relationship and then devising a series of test that will cover the graph so that each object and relationship is exercised and errors are uncovered.
- S.w eng begins by creating graph- collection of nodes that represent objects; links that represent the relationship between objects; node weight that represent properties of node; and link weights that describe some characteristics of link.
- Link may take no of different forms:-

Black Box Testing – Graph Based

FIGURE 17.9
(A) Graph notation
(B) Simple example



GRAPH BASED TESTING

- Transaction Flow Modeling
- Finite State Modeling
- Data Flow Modeling
- Timing Modeling

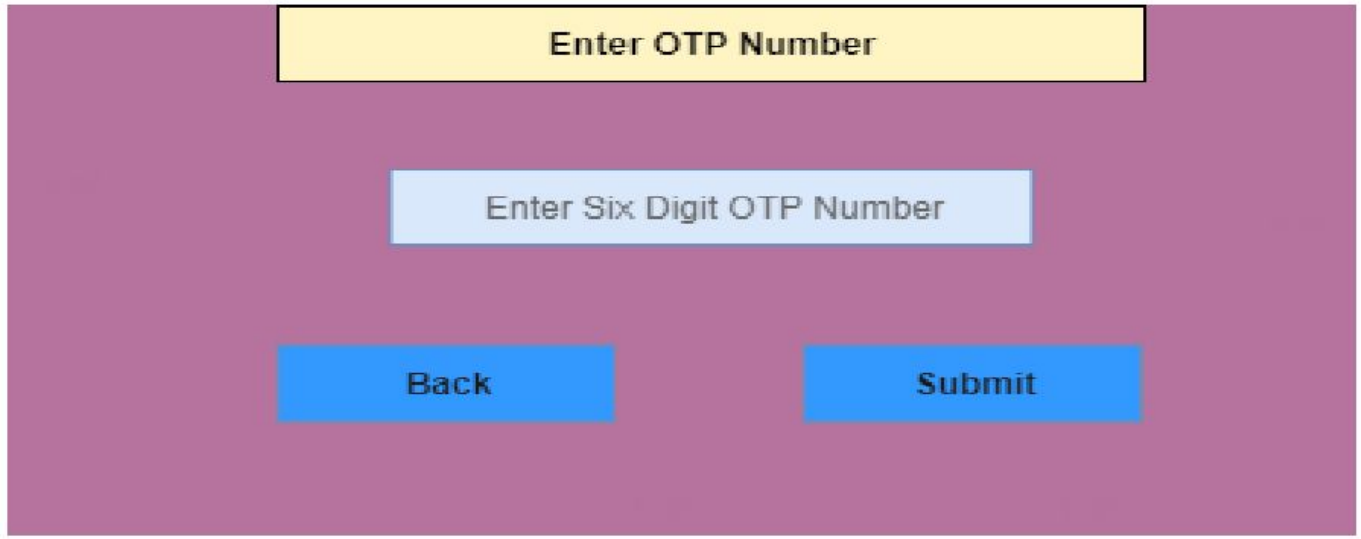
- Transaction flow testing – nodes represent steps in some transaction and links represent logical connections between steps that need to be validated.
- Finite state modeling – nodes represent user observable states of the software and links represent transitions between states.
- Data flow modeling – nodes are data objects and links are transformations from one data object to another.
- Timing modeling – nodes are program objects and links are sequential connections between these objects, link weights are required execution times.

EQUIVALENCE PARTIONING

- Equivalence partitioning is a type of black box testing.
- Divide the input data into equivalence classes/partition.
- Partition represent Valid input values.
- Partition represent Invalid input values.

- Equivalence partitioning is a technique of software testing in which input data is divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behavior.
- If a condition of one partition is true, then the condition of another equal partition must also be true, and if a condition of one partition is false, then the condition of another equal partition must also be false.
- The principle of equivalence partitioning is, test cases should be designed to cover each partition at least once. Each value of every equal partition must exhibit the same behavior as other.

1. OTP Number = 6 digits



INVALID 1 Test case	INVALID 2 Test case	VALID 3 Test case	VALID
DIGITS >=7	DIGITS<=5	DIGITS = 6	DIGITS = 6
93847262	9845	456234	451483

BOUNDARY VALUE ANALYSIS

- Boundary Value Analysis is also called range checking.
- Equivalence partitioning and boundary value analysis are closely related and can be used together at all levels of testing.
- Valid boundaries (in the valid partitions)
- Invalid boundaries (in the invalid partitions)
- 3 Classes:

  i)On the Boundary

  ii) Below the Boundary

  iii) Above the Boundary

BOUNDARY VALUE ANALYSIS Example

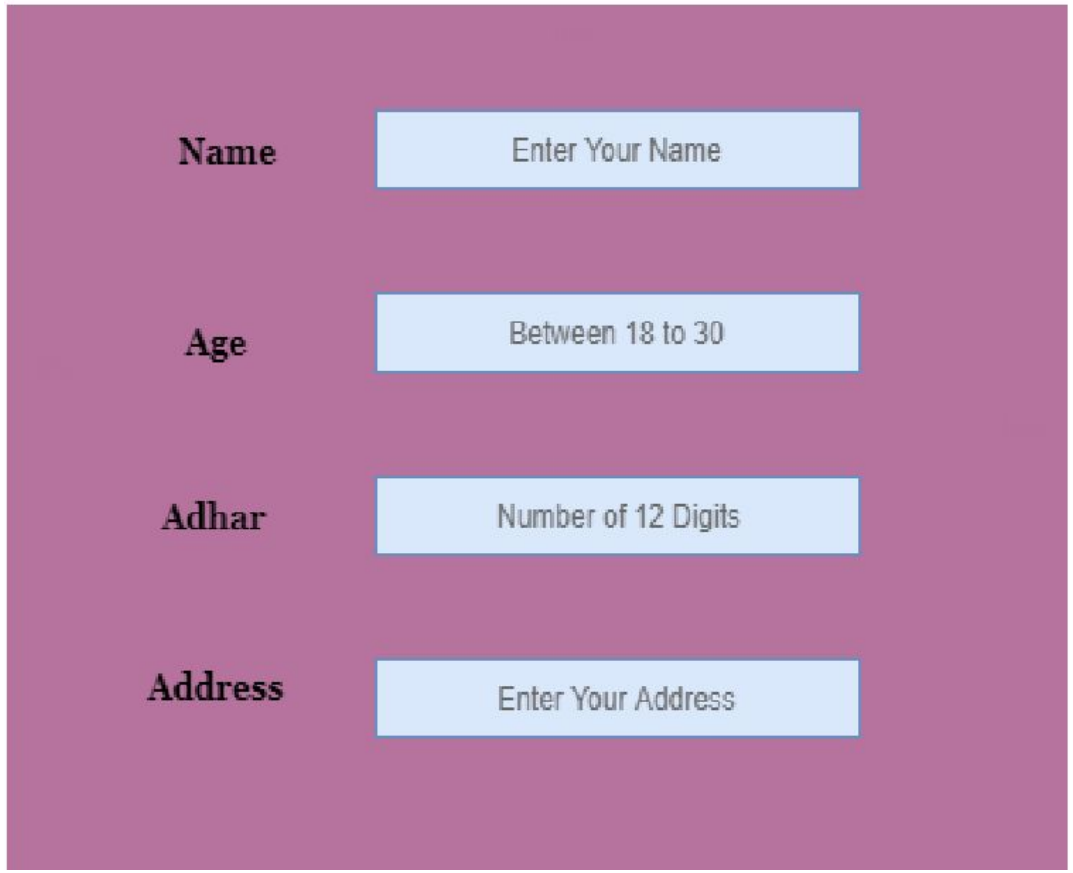
- A program validates a numeric field as follows: values less than 10 are rejected, values between 10 and 21 are accepted, values greater than or equal to 22 are rejected. Which of the following covers the MOST boundary values?
  - Class I: values < 10 => Above the Boundary
  - Class II: 10 to 21 => On the Boundary
  - Class III: values >= 22 => Below the Boundary

- Boundary value analysis is one of the widely used case design technique for black box testing. It is used to test boundary values because the input values near the boundary have higher chances of error.

- Whenever we do the testing by boundary value analysis, the tester focuses on, while entering boundary value whether the software is producing correct output or not.

- Boundary values are those that contain the upper and lower limit of a variable. Assume that, age is a variable of any function, and its minimum value is 18 and the maximum value is 30, both 18 and 30 will be considered as boundary values.

There is 18 and 30 are the boundary values that's why tester pays more attention to these values, but this doesn't mean that the middle values like 19, 20, 21, 27, 29 are ignored. Test cases are developed for each and every value of the range.



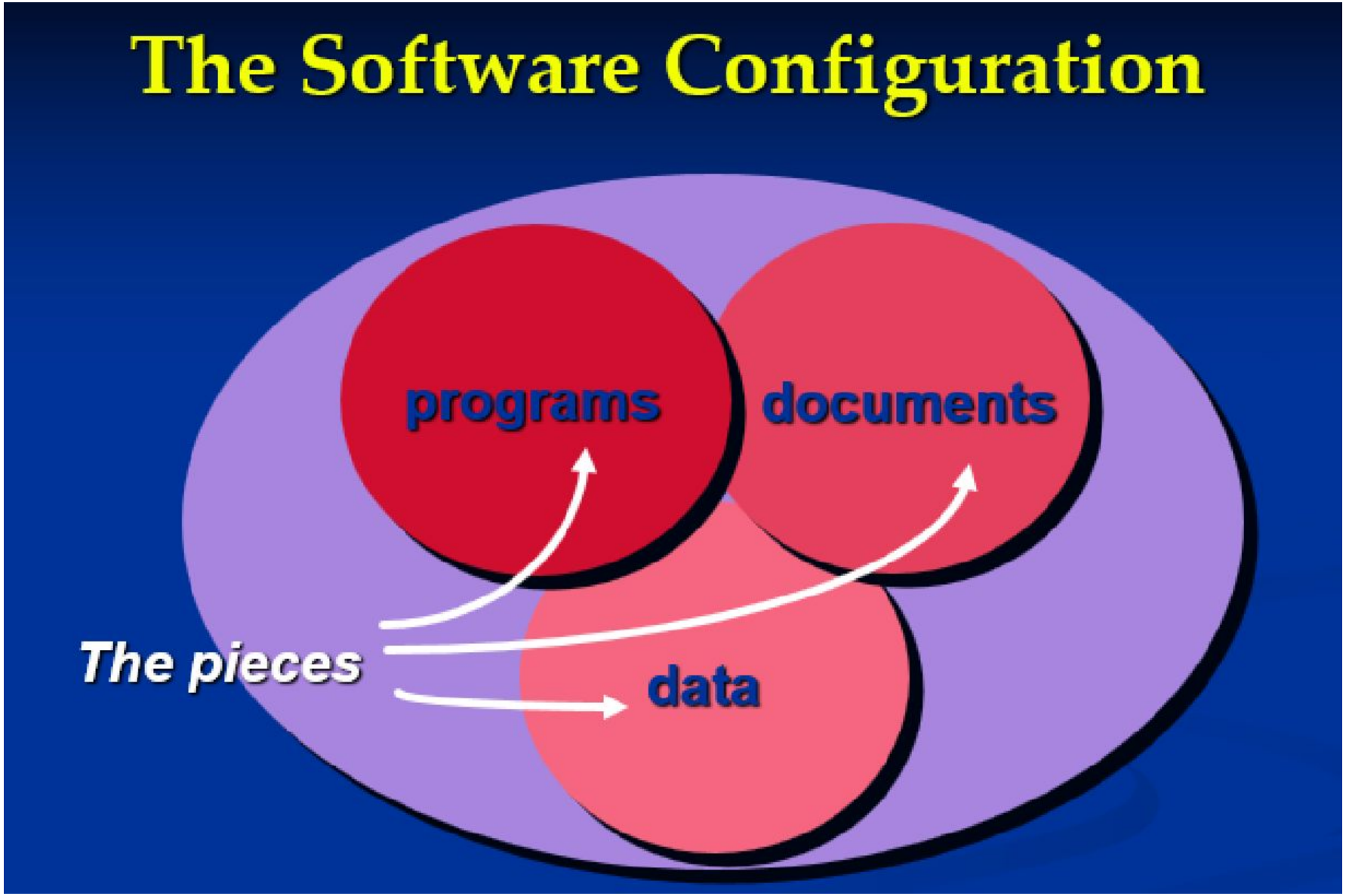
Imagine, there is a function that accepts a number between 18 to 30, where 18 is the minimum and 30 is the maximum value of valid partition, the other values of this partition are 19, 20, 21, 22, 23, 24, 25, 26, 27, 28 and 29. The invalid partition consists of the numbers which are less than 18 such as 12, 14, 15, 16 and 17, and more than 30 such as 31, 32, 34, 36 and 40. Tester develops test cases for both valid and invalid partitions to capture the behavior of the system on different input conditions.

Invalid test cases	Valid test cases	Invalid test cases
11, 13, 14, 15, 16, 17	18, 19, 24, 27, 28, 30	31, 32, 36, 37, 38, 39

Why Orthogonal Array Testing (OATS)..??

- Systematic, statistical way to test pair-wise interactions.
- Interactions and integration points are a major source of defects.
- Most defects arise from simple pair-wise interactions.
  - "When the background is blue and the font is Arial and the layout has menus on the right and the images are large and it's a Thursday then the tables don't line up properly."
- Exhaustive testing is impossible.
- Execute a well-defined, concise set of tests that are likely to uncover most (not all) bugs.
- Orthogonal approach guarantees the pair-wise coverage of all variables.

The Software Configuration



The Software Configuration

Software Process Output is divided into three categories

- Computer Programs (**PROGRAMS**)
- Work products that describe the computer programs (**DOCUMENTS**)
- **Data** (within the program or external to it)

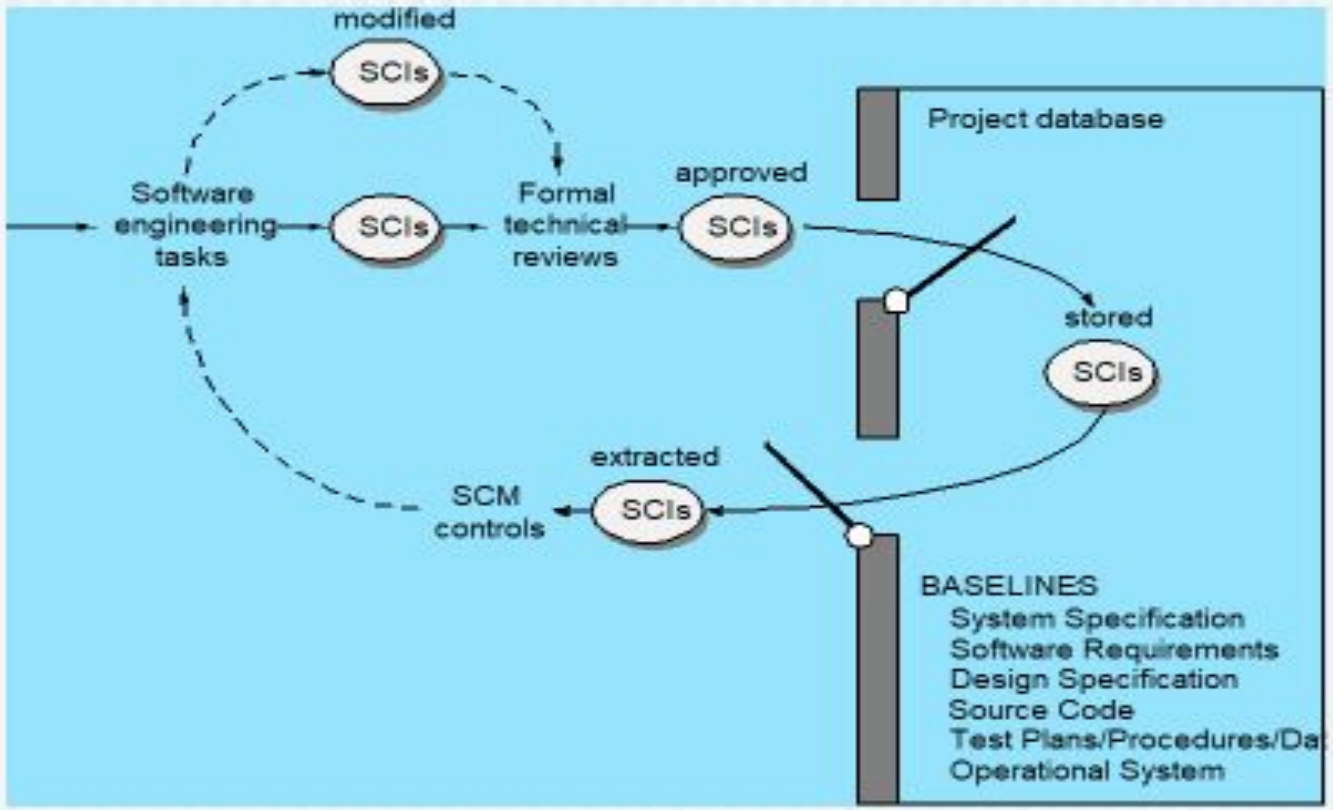
Therefore, SCM is the discipline which

- Identify change
- Monitor and control change
- Ensure the proper implementation of change made to the item.
- Auditing and reporting on the change made.
- Configuration Management (CM) is a technic of identifying, organizing, and controlling modification to software being built by a programming team.
- Multiple people are working on software which is consistently updating. It may be a method where multiple version, branches, authors are involved in a software project, and the team is geographically distributed and works concurrently. It changes in user requirements, and policy, budget, schedules need to be accommodated.

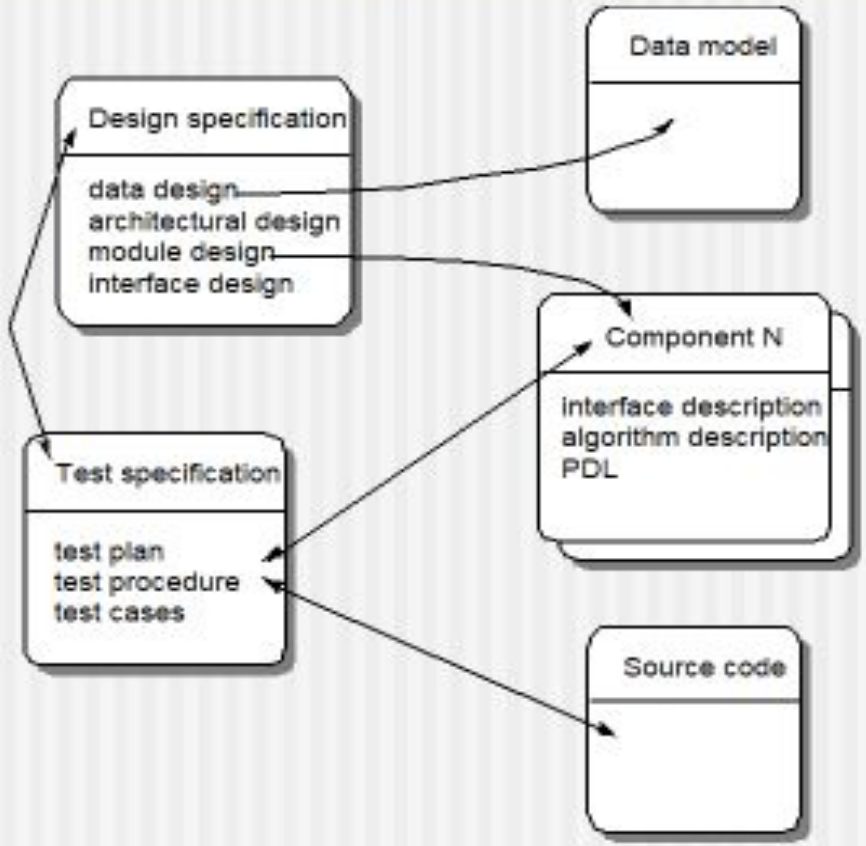
SCM Elements

- *Component elements*—a set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- *Process elements*—a collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering and use of computer software.
- *Construction elements*—a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- *Human elements*—to implement effective SCM, the software team uses a set of tools and process features (encompassing other CM elements)

Baselines



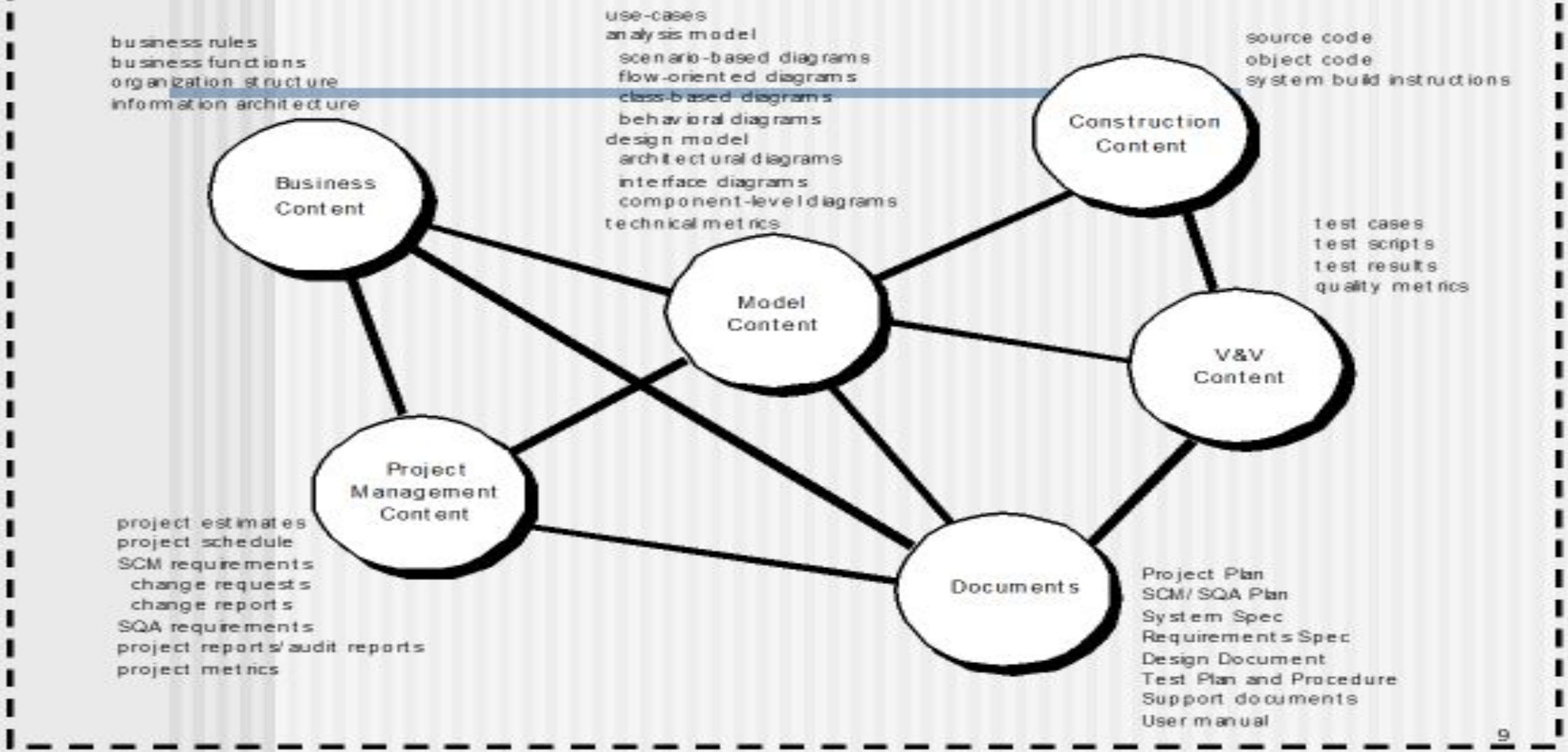
Software Configuration Objects



SCM Repository

- The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner
- The repository performs or precipitates the following functions [For89]:
  - Data integrity
  - Information sharing
  - Tool integration
  - Data integration
  - Methodology enforcement
  - Document standardization

Repository Content



Repository Features

- **Versioning.**
  - saves all of these versions to enable effective management of product releases and to permit developers to go back to previous versions
- **Dependency tracking and change management.**
  - The repository manages a wide variety of relationships among the data elements stored in it.
- **Requirements tracing.**
  - Provides the ability to track all the design and construction components and deliverables that result from a specific requirement specification
- **Configuration management.**
  - Keeps track of a series of configurations representing specific project milestones or production releases. Version management provides the needed versions, and link management keeps track of interdependencies.
- **Audit trails.**
  - establishes additional information about when, why, and by whom changes are made.

The SCM Process

Four primary objectives

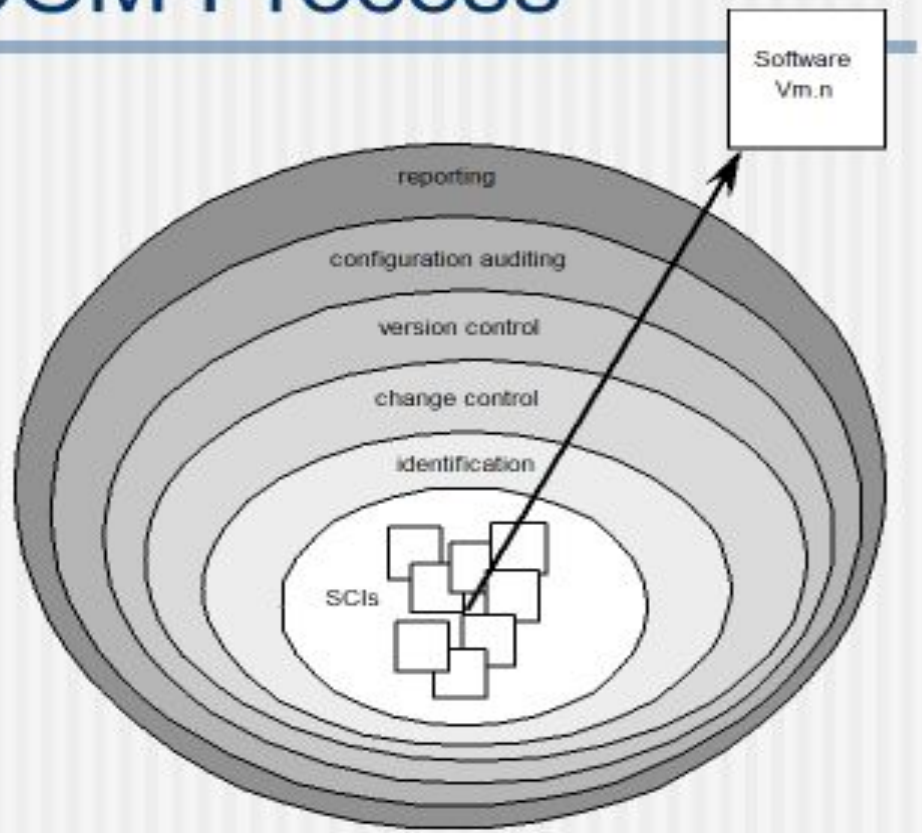
- ✔ To identify all items that collectively define the software configuration
- ✔ To Manage changes to one or more of these items
- ✔ To facilitate the construction of different version of an application
- ✔ To ensure that software quality is maintained as the configuration evolves over time

The SCM Process

Addresses the following questions …

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?

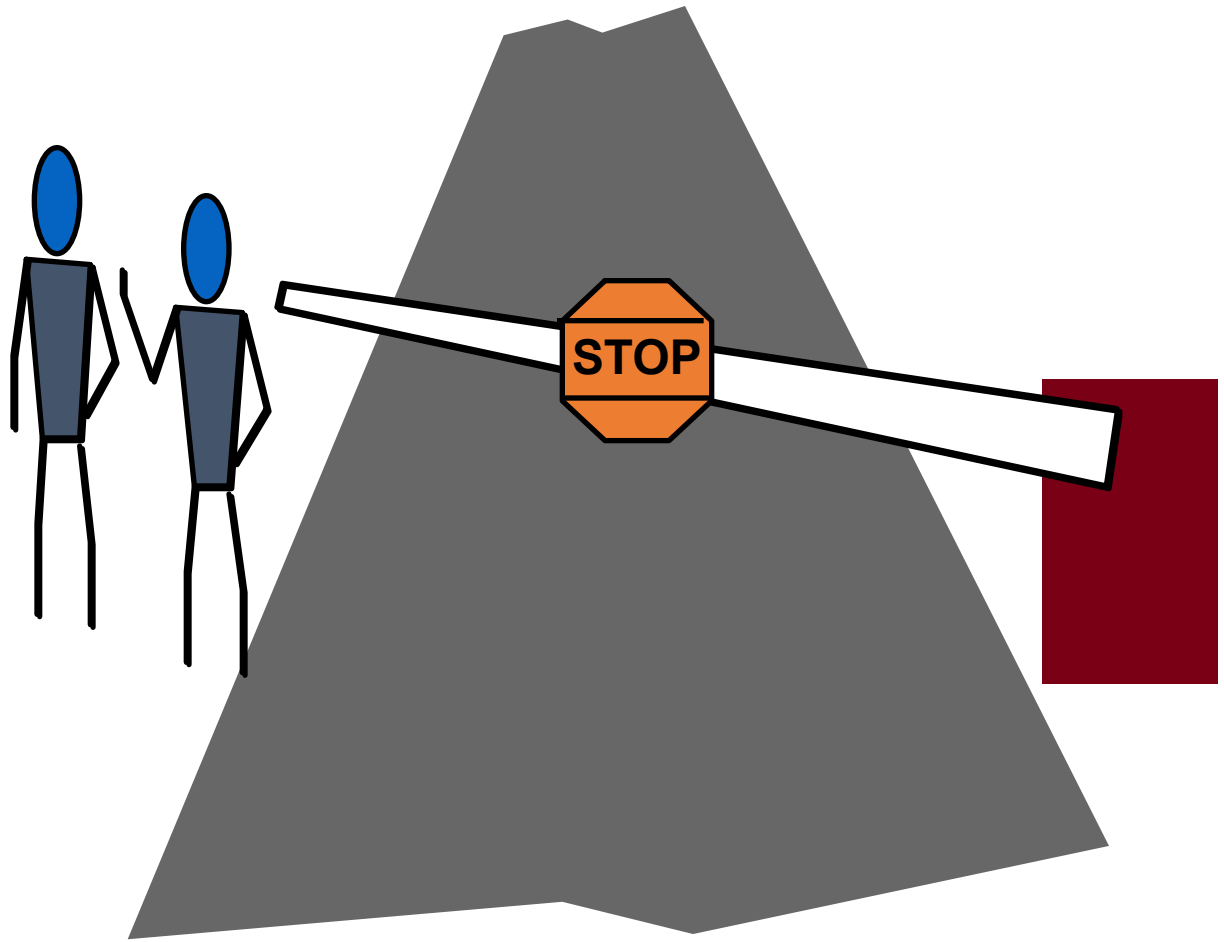
The SCM Process



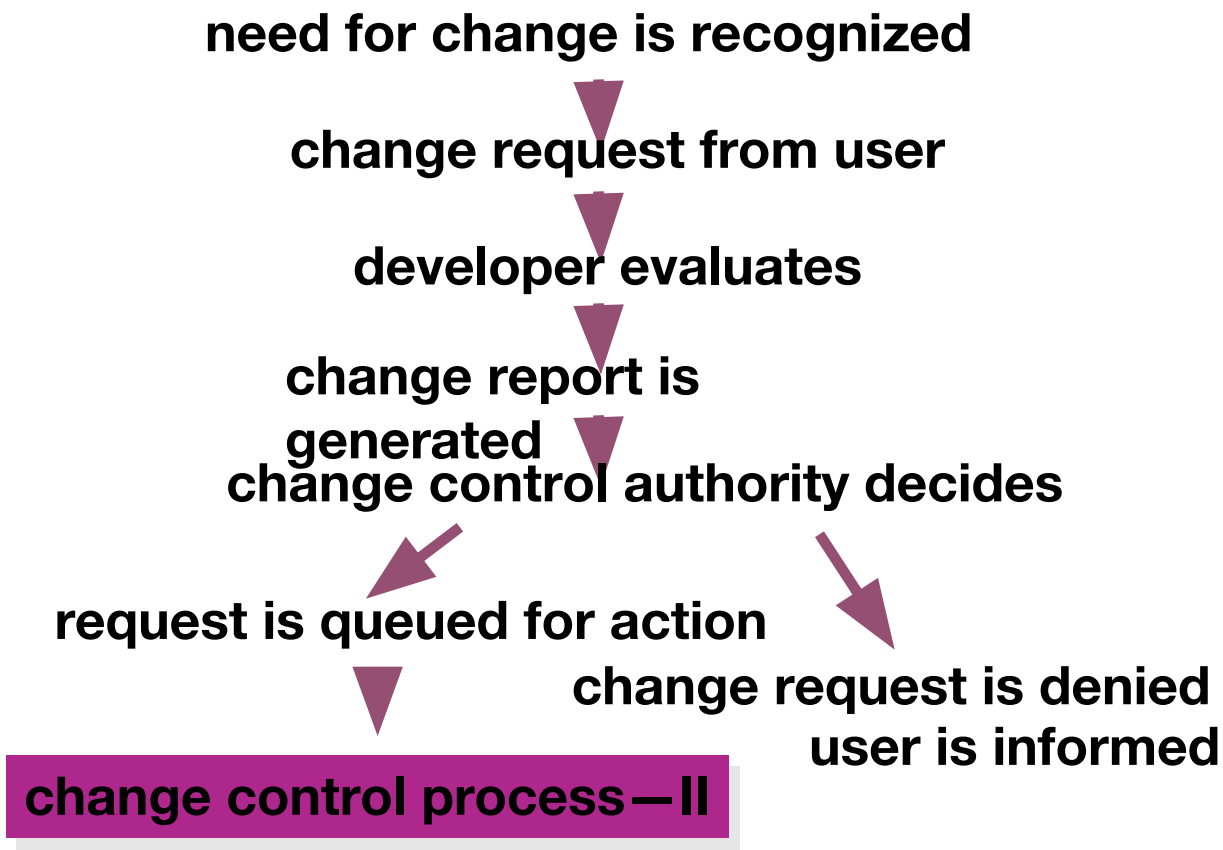
Version Control

- Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process
- A version control system implements or is directly integrated with four major capabilities:
  - a *project database (repository)* that stores all relevant configuration objects
  - a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions);
  - a *make facility* that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software.
  - an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

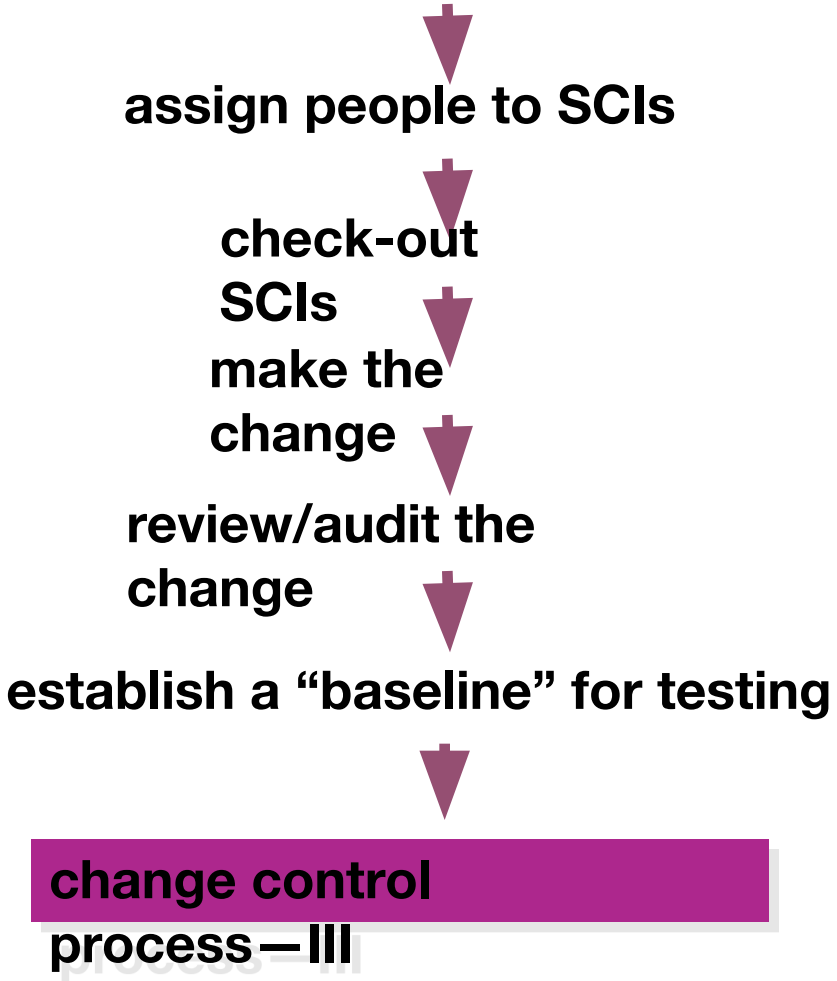
Change Control



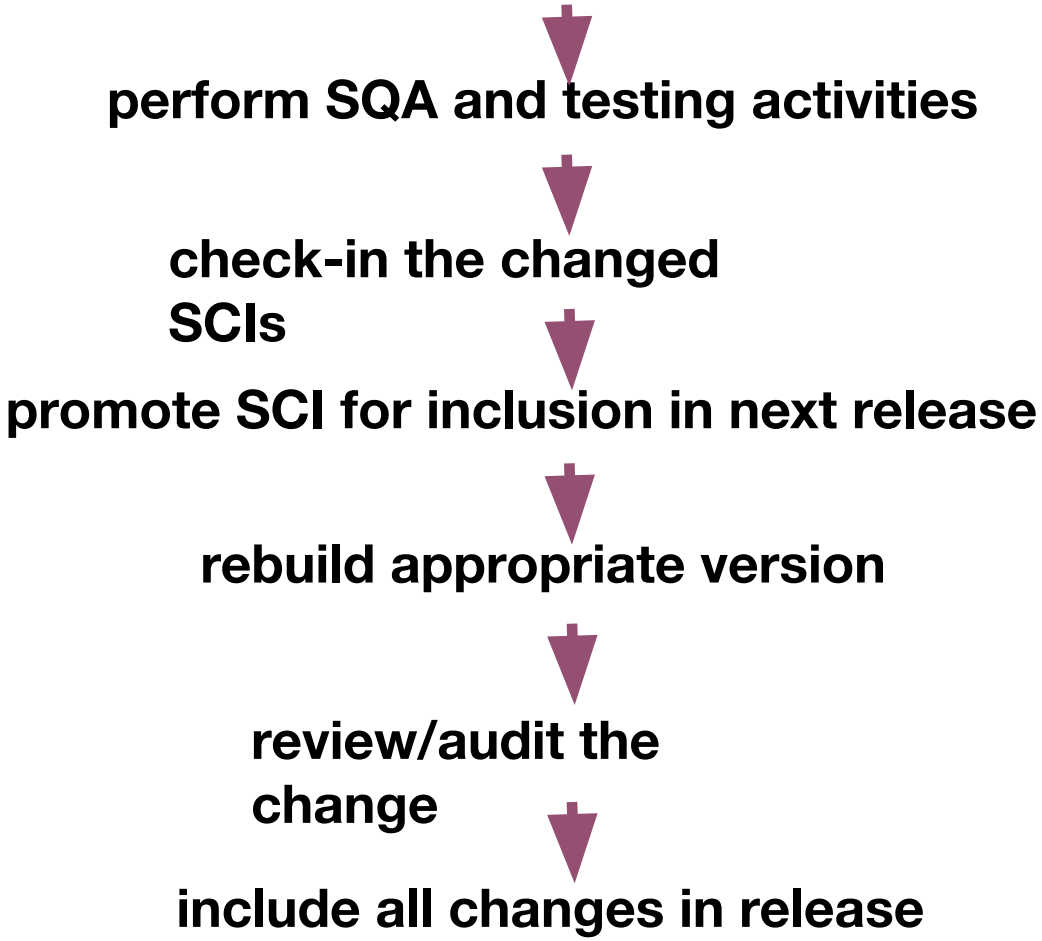
Change Control Process—I



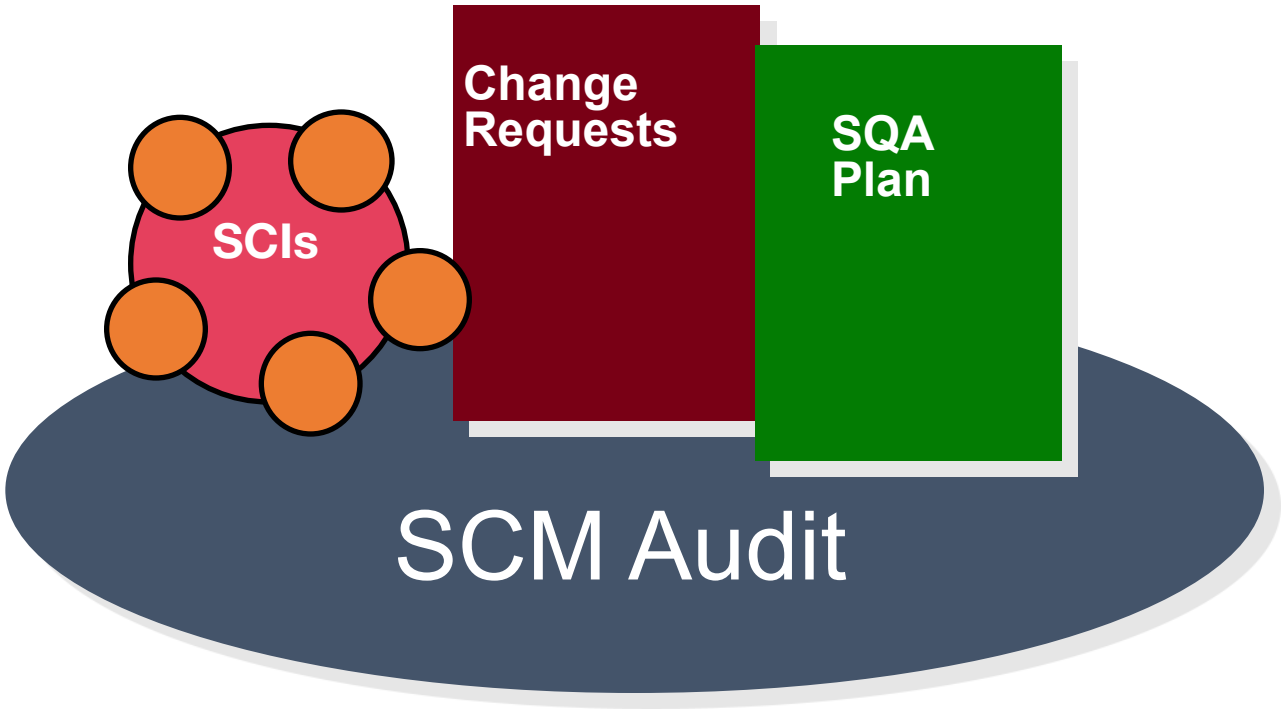
Change Control Process-II



Change Control Process-III



Auditing



Status Accounting