# Quality Concepts

# What Is Quality?

- David Garvin: Quality is a ***complex and multifaceted concept*** that can be described *from five different points of view.*

- The transcendental view argues that you immediately recognize, but can not explicitly define.
- The user view sees quality in terms of an end user's specific goals. If the product meets those goals, it exhibits quality.
- The manufacturer's view defines quality in terms of the original specification of the product. If the product conforms the specification, it exhibits quality.
- The product view suggests that quality can be tied to inherent characteristics of the product.

- The value-based view measures quality based on how much a customer is willing to pay for a product.

- **Quality of design** encompasses the degree to which the designer meets the functions and features specified in the requirement model.
- **Quality of conformance** focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

# Software Quality

Software quality can be defined as :

- “An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce and who those use it.”

 The definition serves to emphasize three important points :

1. An effective software process establishes the infrastructure that supports any effort at building a high-quality software product.
2. A useful product delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way. A useful product always satisfies those requirements that have been explicitly stated by stake holders. It satisfies a set of implicit requirements that are expected of all high-quality software.

3. By adding value for both the producer and user of a software product, high-quality software provides benefits for the software organization and the end-user community. The software organization gains added value because high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.

   This enables software engineers to spend more time creating new applications and less on rework.

   The user community gains added value because the application provides a useful capability in a way that expedites some business process.

   The end result is

   (1) greater software product revenue,

   (2) better profitability when an application supports a business process, and/or

   (3) improved availability of information that is crucial for the business.

# Garvin's Quality Dimensions

- David Garvin suggests that quality should be considered by taking a multidimensional viewpoint that begins with an assessment of conformance and terminates with a transcendental view.
- Following dimensions of quality can be applied when software quality is considered:

 **Performance Quality :** Does the software delivers all content, functions, and features that are specified as part of requirements model in a way that provides value to the end user?

- **Feature Quality** : Does the software provide features that surprise and delight first-time end users?
- **Reliability :** Does the software deliver all features and capability without failure? Is it available when it is needed?
- **Conformance :** Does the software conform to local and external software standards that are relevant to the applications? E.g. does the user interface conform to accepted design rules for menu selection or data input?
- **Durability :** Can software be maintained or corrected without the inadvertent generation of side effects? Will change causes the error rate or reliability to degrade with time?

- **Serviceability :** Can the software be maintained or corrected in an acceptably short time?
- **Aesthetics** : There's no question that each of us has a different and very subjective vision of what is aesthetic. And yet, most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and a obvious "presence" that are hard to quantify but are evident (clear) nonetheless.
- **Perception** : In some situation, you have a set of prejudices that will influence your perception of quality. Apart from this, we need a set of "hard" quality factors that can be categorized into two groups : (1) factors that can be directly measured (e.g. defects uncovered during testing) (2) factors that can be measured indirectly (e.g. usability)

# McCall's Quality Factors

- Figure 14.1 Software quality factors of figure focus on three important aspect of a software product:-

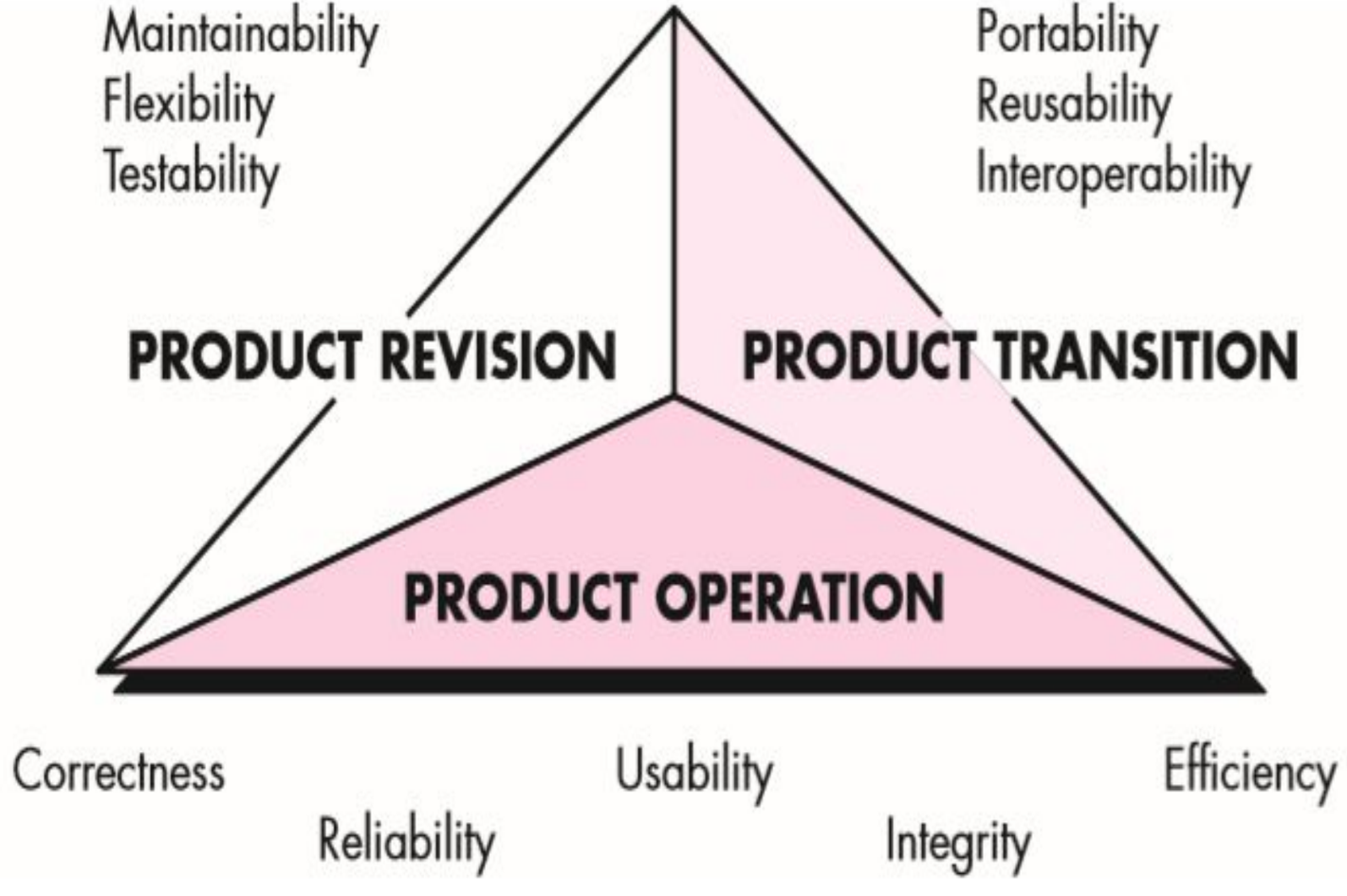
(1)Its operational characteristics

(2)Its ability to undergo changes

(3)Adaptability to new environment

**FIGURE 14.1**

McCall's software quality factors



- **Correctness**:- the extent to which a program satisfies its specification and fulfills the customer's mission objectives.
- **Reliability** :- the extent to which a program can be expected to perform its intended function with required care.
- **Efficiency** :- the amount of computing resources and code required by a program to perform its function.
- **Integrity:**- the extent to which access to software or data by unauthorized persons can be controlled.
- **Usability:**- the effort required to learn, operate, prepare input for and interpret output of a program.

- **Maintainability:**- the effort required to locate error in the program.
- **Flexibility:**- the effort required to modify an operational program.
- **Testability:**- the effort required to test a program to ensure that it performs its intended function

- **Portability**:- the effort required to transfer the program from one hardware and/or software system environment to another.
- **Reusability:**- the extent to which a program [or parts of program] can be reused in another applications.
- **Interoperability:**- the effort required to couple one system to another.

# ISO 9126 Quality Factors

- It was developed in attempt to identify quality attributes for computer software. The standard identifies six key quality attributes.

(1)**Functionality**:- the degree to which the software satisfies needs as indicated by the following sub-attributes: accuracy, suitability, interoperability etc.

(2)**Reliability:**- The amount of time that the software is available for use as indicated by the flowing sub attributes : fault tolerance, recoverability

(3)**Usability**:- the degree to which software is easy to use as indicated by the these attributes: understandability, learn ability etc

(4)**Efficiency**:- the degree to which the software makes optimal use of resources as indicated by the following sub attributes: time behavior, resource behavior.

(5) **Maintainability:**- the ease with which repair may be made to the software as indicated by the following attributes: changeability, testability, analyzability etc.

(6) **Portability**: the ease with which the software can be transposed from one environment to another as indicated by these attributes: adaptability, replaceability.

# Targeted Quality Factors

- A software team can develop a set of quality characteristics and associated questions that would probe (check out) the degree to which each factor has been satisfied.
- **Intuitiveness :** The degree to which the interface follows expected usage patterns so that even a novice can use it without significant training. Does the interface follow the three golden rules? Are the interface operations easy to locate and initiate?

- **Efficiency :** The degree to which operations and information can be located or initiated. 1) Are output data or content presented so that it is understood immediately? 2) Does the interface layout and style allow a user to locate operations and information efficiently?
- **Robustness :** The degree to which the software handles bad input data or inappropriate user interaction. 1) Does the interface provides useful diagnosis and guidance when error condition is uncovered? 2) Will the interface recognize common manipulative mistakes and explicitly guide the user back on the right back?

- **Richness :** The degree to which the interface provides a rich feature set. 1) Can the interface be customized to the specific needs of a user? 2) Does the interface provide a macro capability that enables a user to identify a sequence of common operations with a single action or command?

# Software Testing Strategies

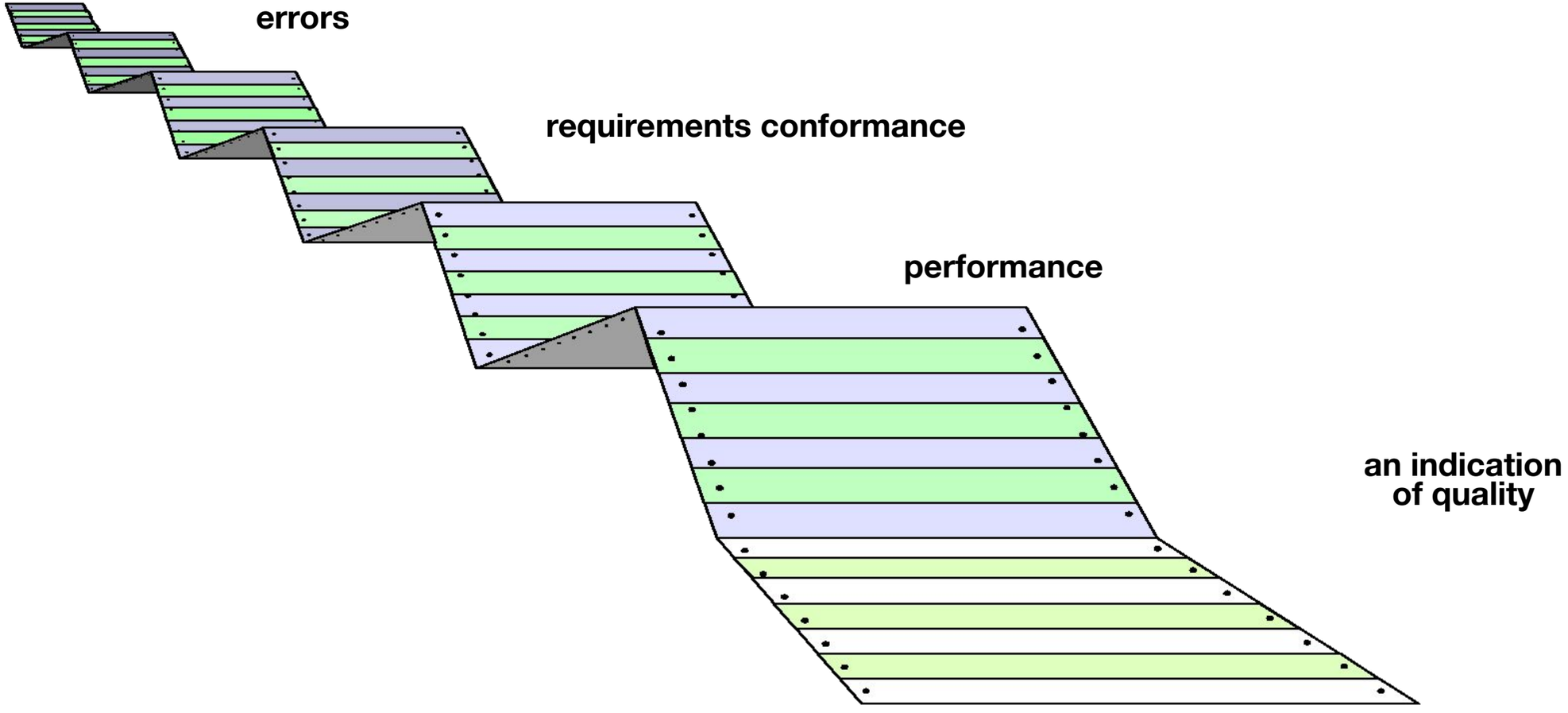
# Software Testing  Strategies

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

- A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required.

- Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.
- A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to encourage reasonable planning and management tracking as the project progresses.

# 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.

# Verification and Validation

- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

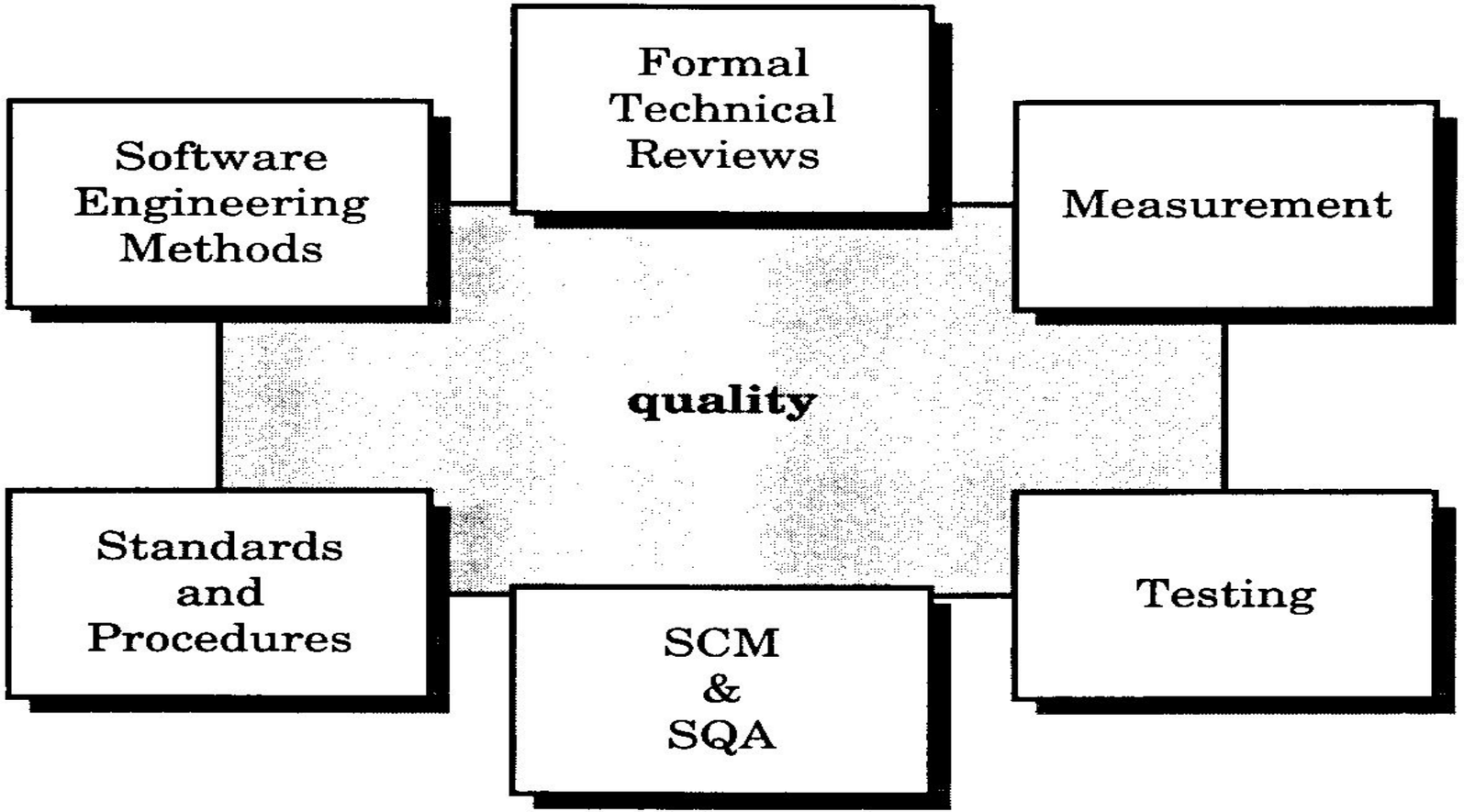



**FIGURE 17.1.**
Achieving software quality

# Organizing for Software Testing

- Testing should aim at "breaking" the software
- Common misconceptions:

  The developer of software should do no testing at all.

  The software should be given to a secret team of testers who will test it unmercifully .

  The testers get involved with the project only when the testing steps are about to begin

- The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed.
- In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture.
- Only after the software architecture is complete does an independent test group become involved.
- The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present.

- ITG personnel are paid to find errors. However, don’t turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted.
- (“The first mistake that people make is thinking that the testing team is responsible for assuring quality.”- Brian Marick)
- While testing is conducted, the developer must be available to correct errors that are uncovered.

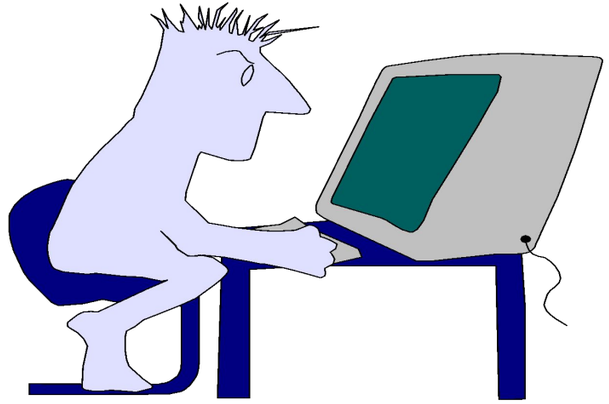
# Who Tests the Software?

*developer*

**Understands the system**

**but, will test "gently"**

**and, is driven by "delivery"**



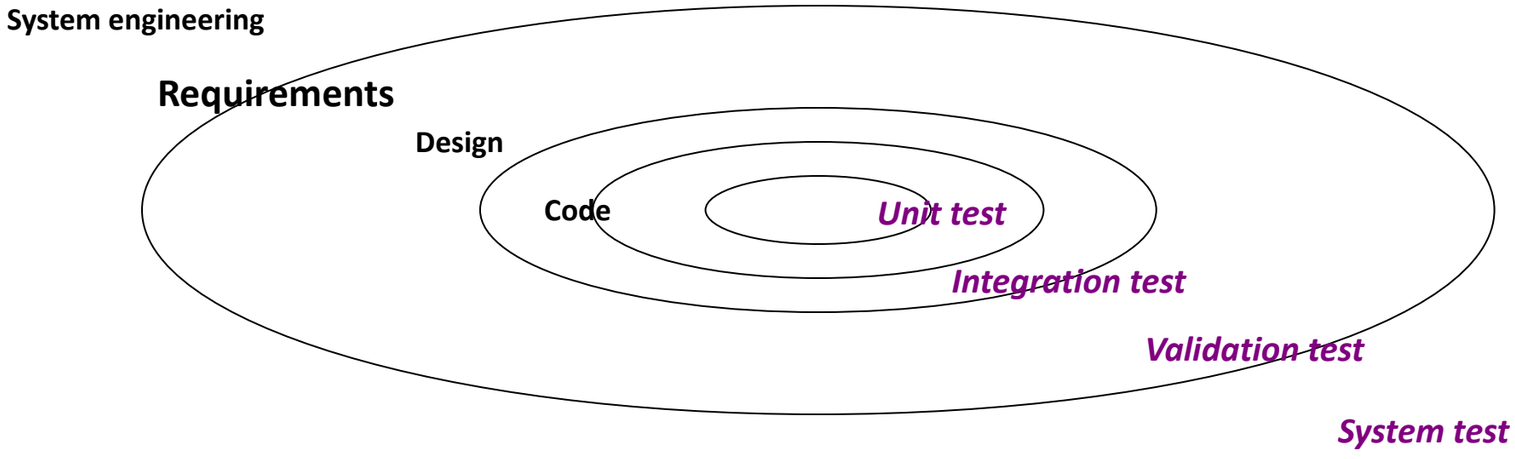
***independent tester***

**Must learn about the system,**

**but, will attempt to break it**

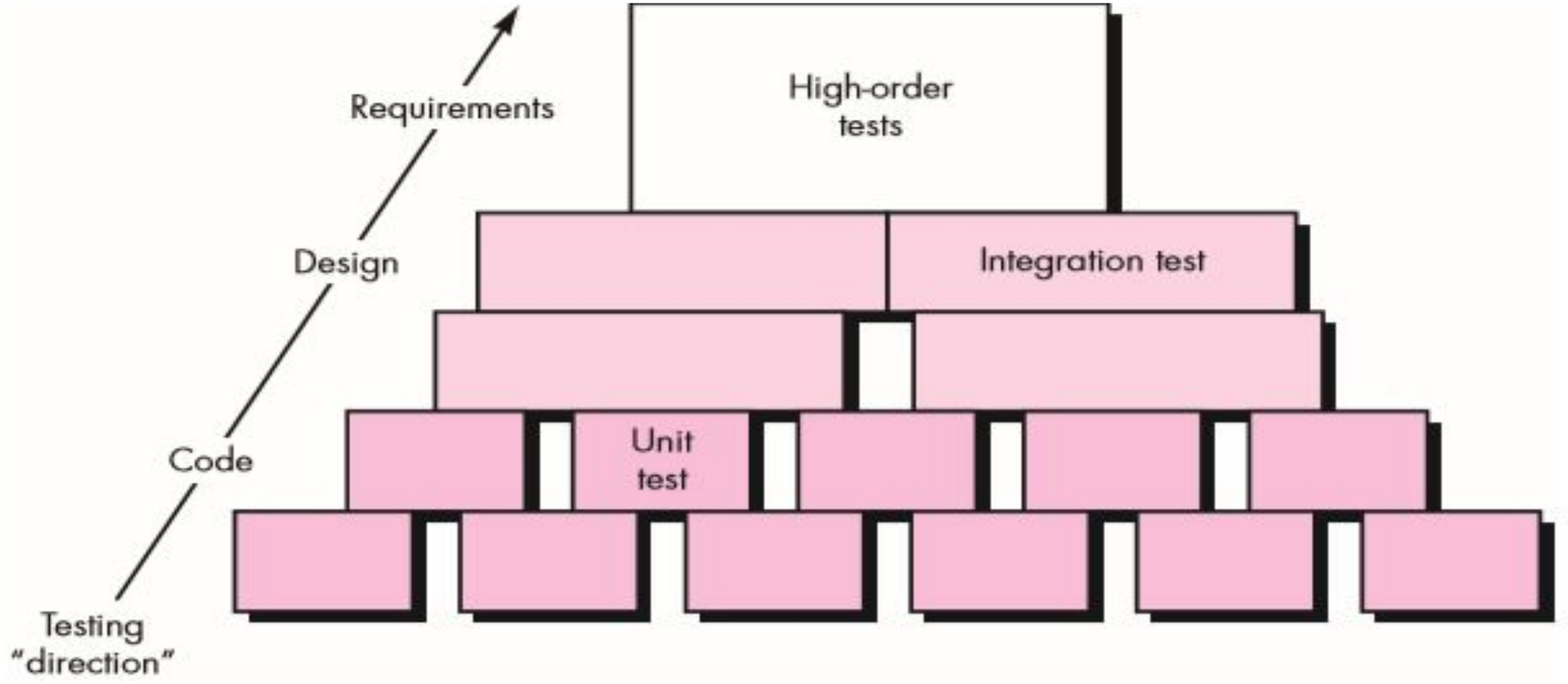
**and, is driven by quality**

# Software 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

# Procedural point of view



# 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

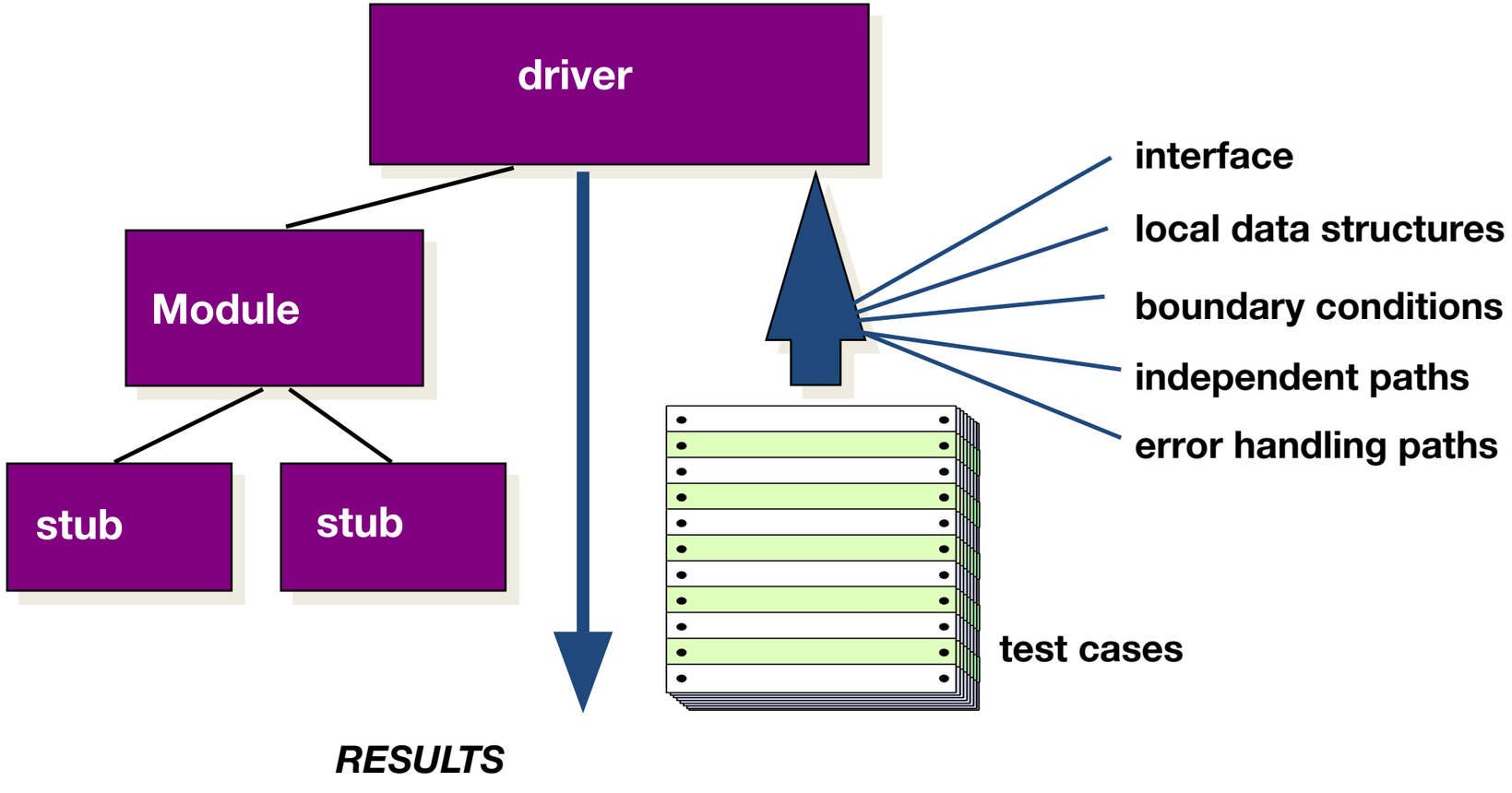
# Unit Testing

- Concentrates on each component/function of the software
- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited
- Targets for Unit Test Cases
  - Module interface
  - Local data structures
  - Boundary conditions
  - Independent paths (basis paths)
  - Error handling paths

# Drivers and Stubs for Unit Testing

- Driver
  - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- Stubs
  - Serve to replace modules that are subordinate to (called by) the component to be tested
  - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- Drivers and stubs both represent overhead
  - Both must be written but don't constitute part of the installed software product

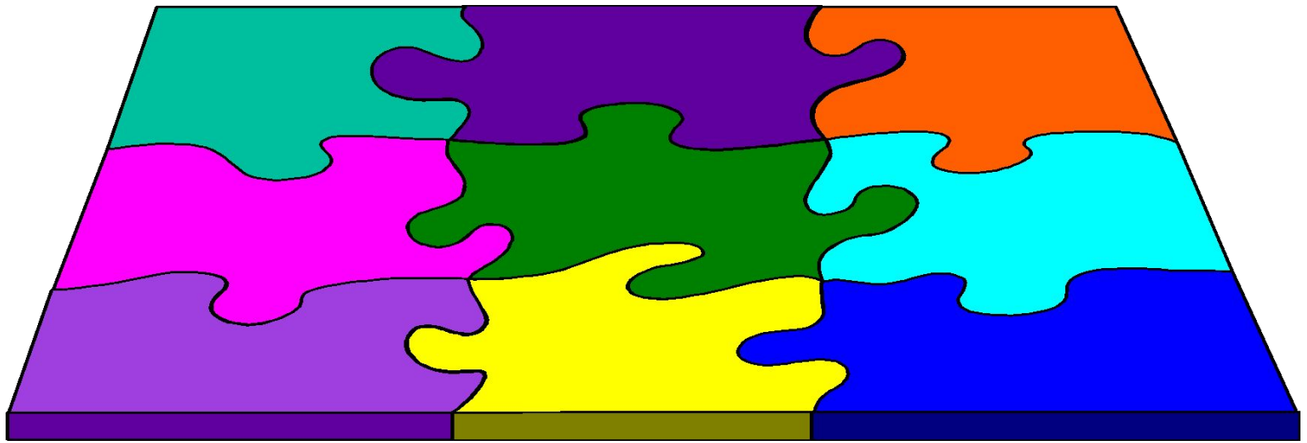
# Unit Test Environment



# Integration Testing Strategies

**Options:**

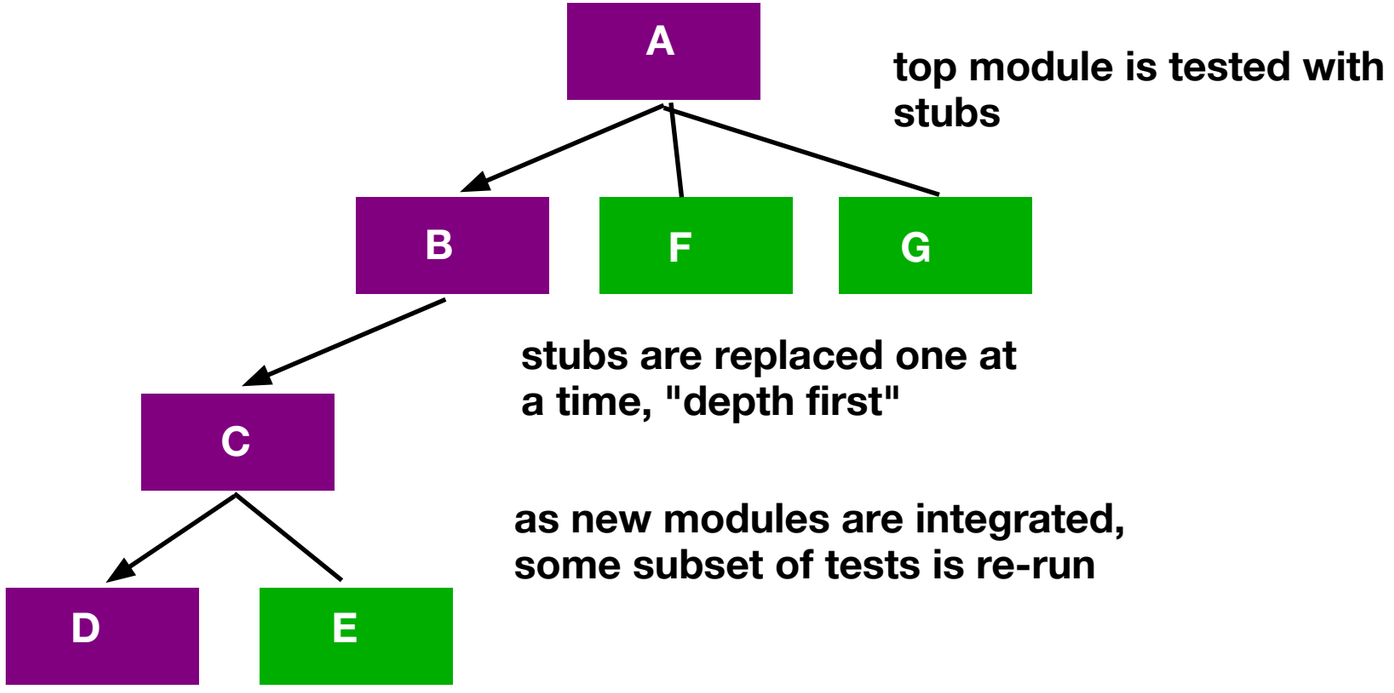
- **the “big bang” approach**
- **an incremental construction strategy**



# Integration Testing

- Integration testing
  - Focuses on the design and construction of the software architecture
  - Focuses on how well the components fit together and work together
  - conduct tests to uncover errors associated with interfaces
- Defined as a systematic technique for constructing the software architecture
- Objective is to take unit tested modules and build a program structure based on the prescribed design

# Top Down Integration



# Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

- The integration process is performed in a series of five steps:

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

# Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy

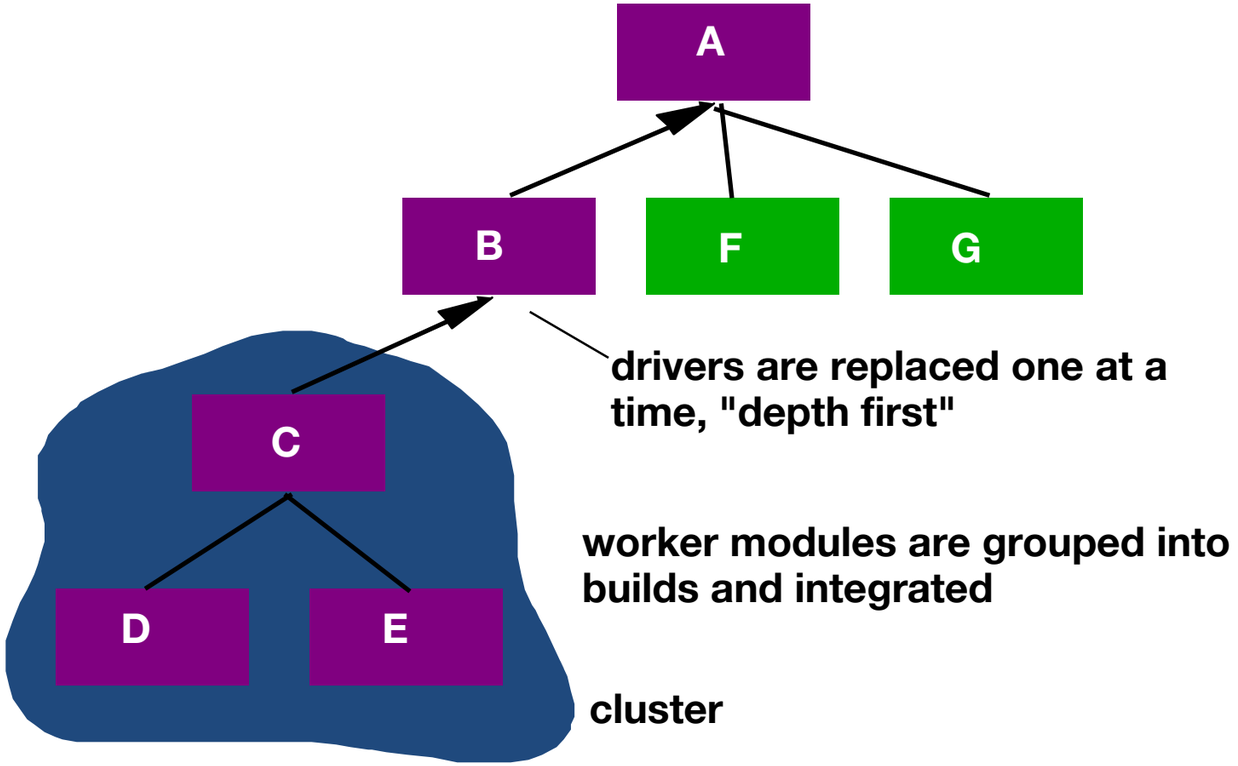
- Advantages
  - This approach verifies low-level data processing early in the testing process
  - Need for stubs is eliminated

- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

- A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

# Bottom-Up Integration



# Regression Testing

- Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked.
- Each new addition or change to baselined software 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 unintended side effects.
- May be done manually or through the use of automated capture/playback tools

# Regression Testing

- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce accidental behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools

- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

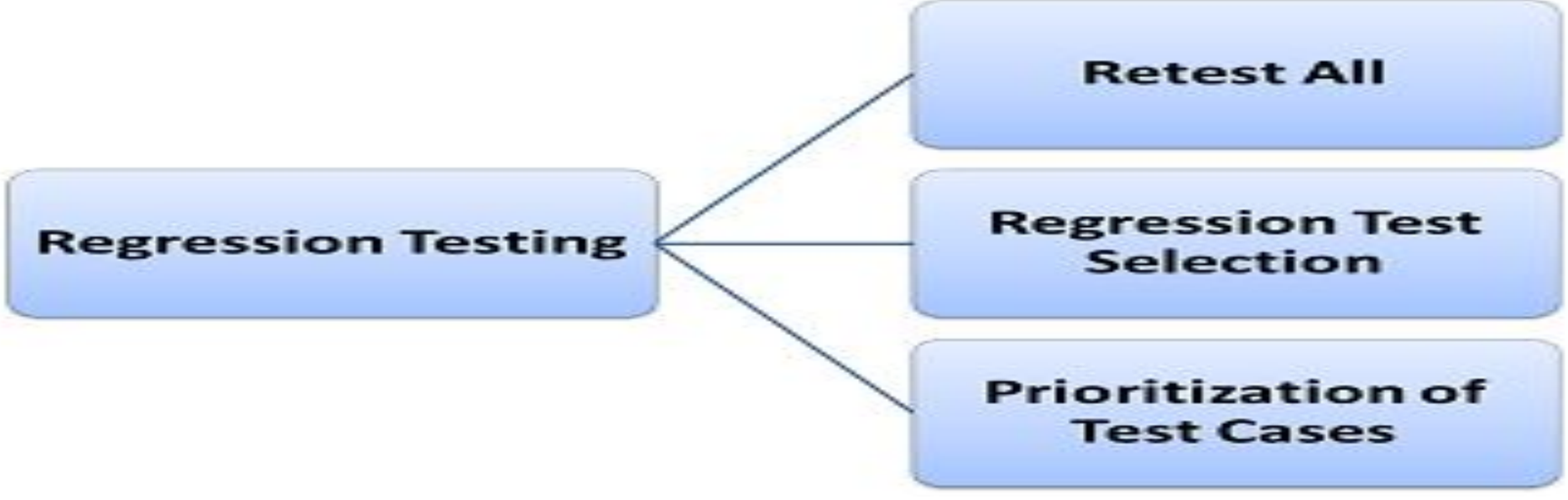
- As integration testing proceeds, the number of regression tests can grow quite large.

Regression Testing is required when there is a

- Change in requirements and code is modified according to the requirement
- New feature is added to the software
- Defect fixing
- Performance issue fix

## Retest All

- This is one of the methods for Regression Testing in which all the tests in the existing test bucket or suite should be re-executed. This is very expensive as it requires huge time and resources.



**Regression Test Selection**

- Instead of re-executing the entire test suite, it is better to select part of the test suite to be run
- Test cases selected can be categorized as 1) Reusable Test Cases 2) Obsolete Test Cases.
- Re-usable Test cases can be used in succeeding regression cycles.
- Obsolete Test Cases can't be used in succeeding cycles.

**Prioritization of Test Cases**

- Prioritize the test cases depending on business impact, critical & frequently used functionalities. Selection of test cases based on priority will greatly reduce the regression test suite.

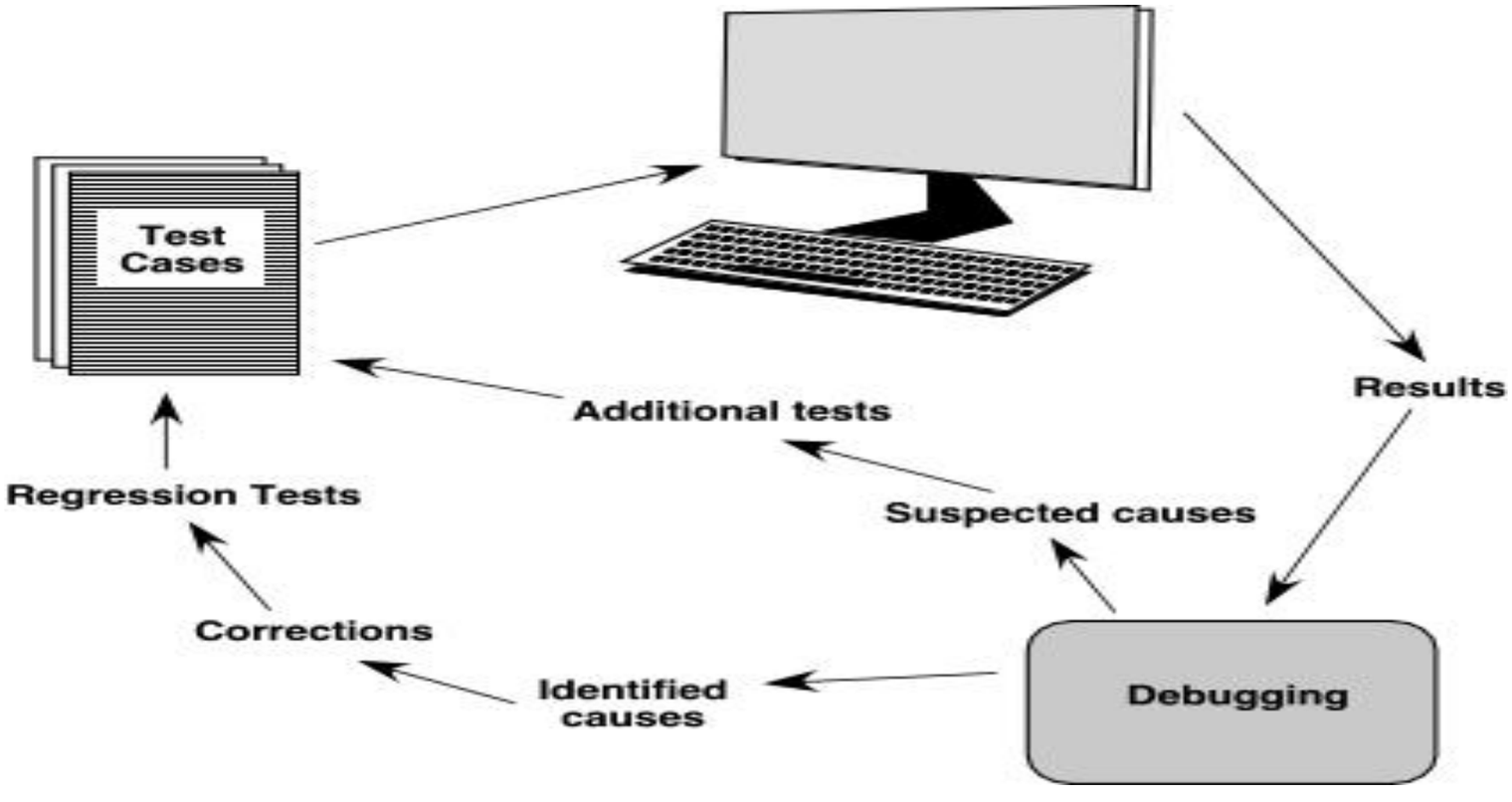
# When is Testing Complete?

- There is no definitive answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various severity levels
  - Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

END of Testing

| Testing | Debugging |
|---|---|
| 1. Testing always starts with known conditions, uses predefined methods, and has predictable outcomes too. | 1. Debugging starts from possibly un-known initial conditions and its end cannot be predicted, apart from statistically. |
| 2. Testing can and should definitely be planned, designed, and scheduled. | 2. The procedures for, and period of, debugging cannot be so constrained. |
| 3. It proves a programmers failure. | 3. It is the programmer's vindication. |
| 4. It is a demonstration of error or apparent correctness. | 4. It is always treated as a deductive process. |
| 5. Testing as executed should strive to be predictable, dull, constrained, rigid, and inhuman. | 5. Debugging demands intuitive leaps, conjectures, experimentation, and some freedom also. |
| 6. Much of the testing can be done without design knowledge. | 6. Debugging is impossible without detailed design knowledge. |
| 7. It can often be done by an outsider. | 7. It must be done by an insider. |
| 8. Much of test execution and design can be automated. | 8. Automated debugging is still a dream for programmers. |
| 9. Testing purpose is to find bug. | 9. Debugging purpose is to find cause of bug. |

# Debugging Process



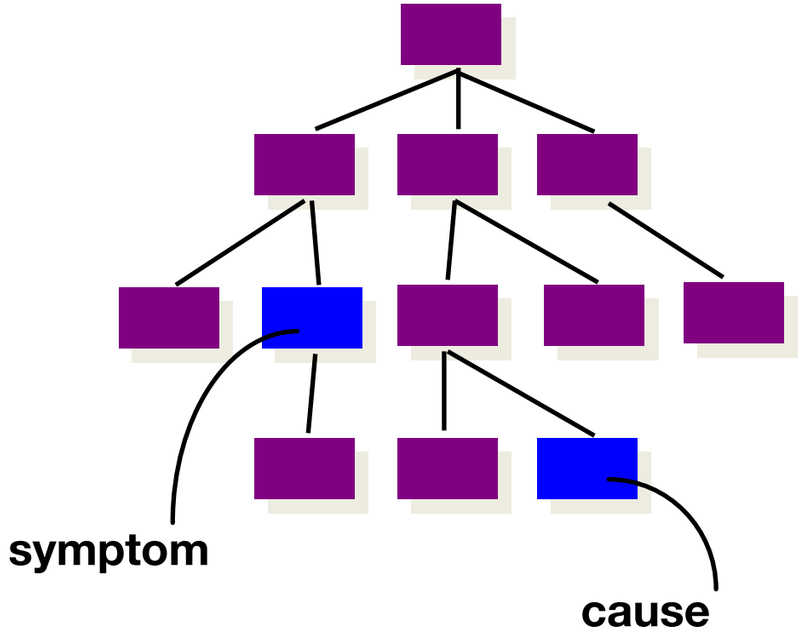
- Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art.
- The debugging process will usually have one of two outcomes:

(1) the cause will be found and corrected or

(2) the cause will not be found.

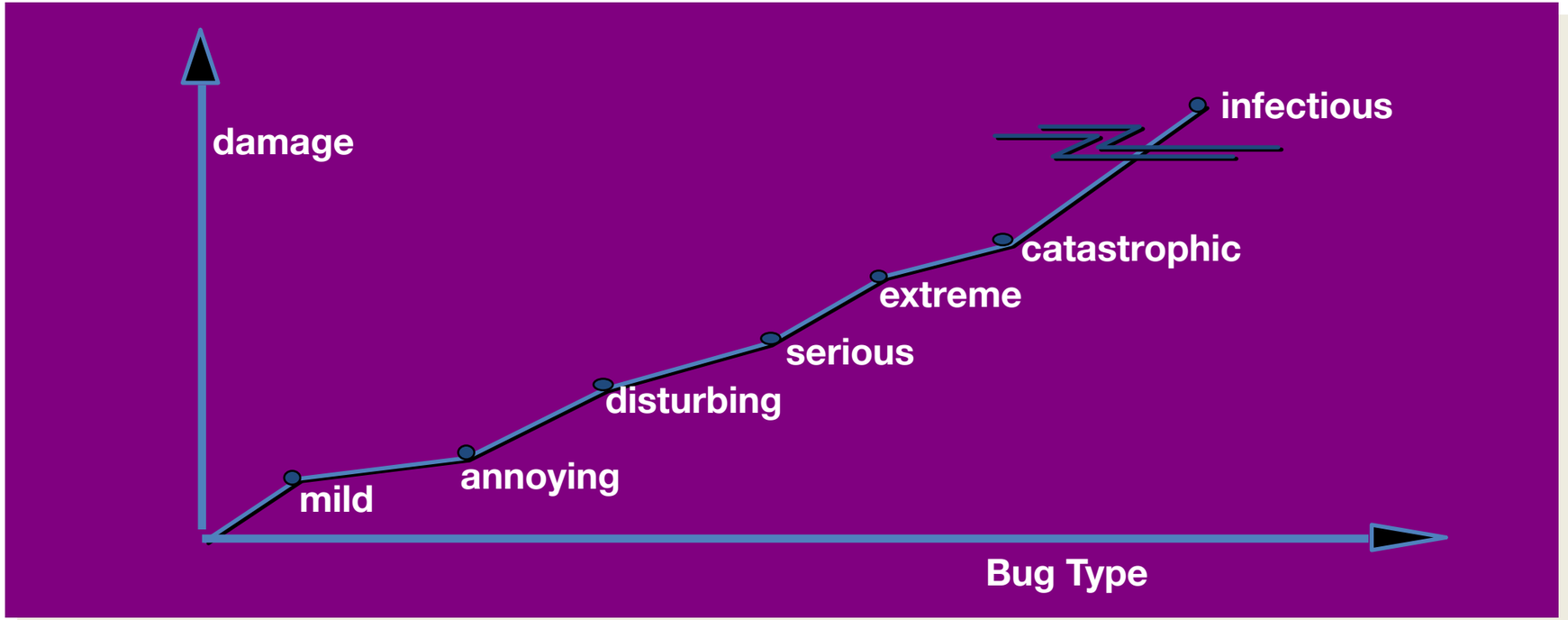
In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

# Symptoms & Causes




- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

# Consequences of Bugs




***Bug Categories:*** **function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.**

# Debugging Strategies

- brute force / testing
- backtracking
- induction
- deduction

# Highlights

- *Think* -- before you act to correct
- Use tools to gain additional insight
- If you're at an impasse, get help from someone else
- Once you correct the bug, use regression testing to uncover any side effects

# Testing Conventional Applications

# Software Testing Fundamentals

- **Testability:** "Software testability is simply how easily [a computer program] can be tested."

  - James Bach

***The following characteristics lead to testable software:***

- 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.

# Test Characteristics

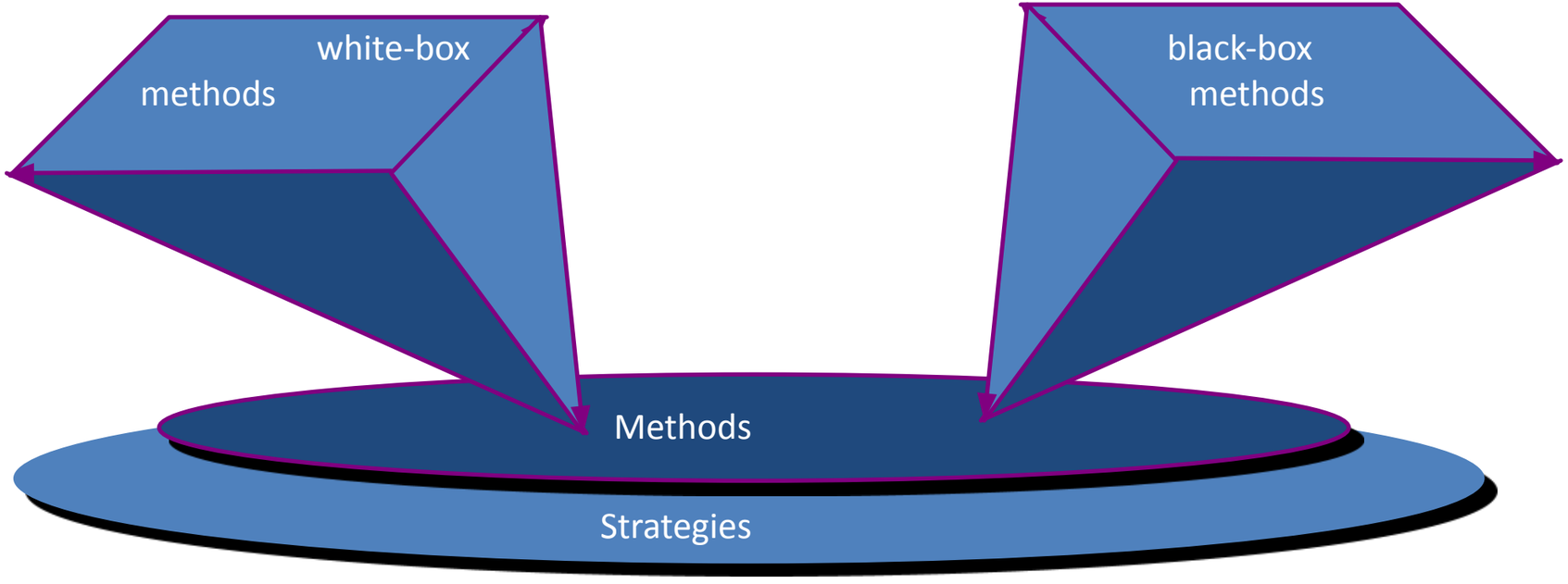
- **A good test has a high probability of finding an error**
  - The tester must understand the software and how it might fail
- **A good test is not redundant**
  - Testing time is limited;
- **A good test should be "best of breed"**
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- **A good test should be neither too simple nor too complex**
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

# Internal and External Views

- Any engineered product (and most other things) can be tested in one of two ways:

- Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
- Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

# Software Testing



# Two Unit Testing Techniques

- **Black-box testing**
  - test to see that function is fully operational & error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software
- **White-box testing**
  - Tests all internal components have been exercised
  - Tests procedural detail
  - Logical paths are tested
  - Test cases exercise specific sets of conditions and loops

# White-box Testing

- White-box testing technique proposed by Tom McCabe
- Uses the control structure part of component-level design to derive the test cases
- These test cases
  - 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

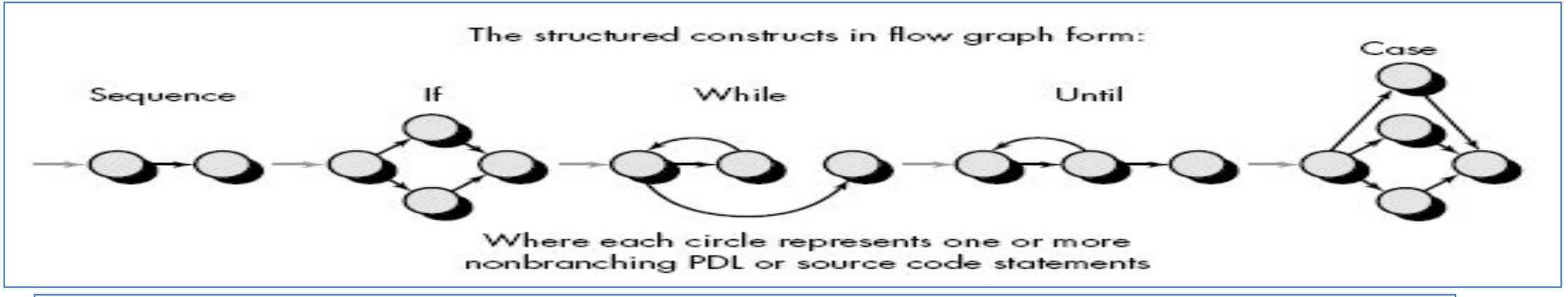
## Basis Path Testing

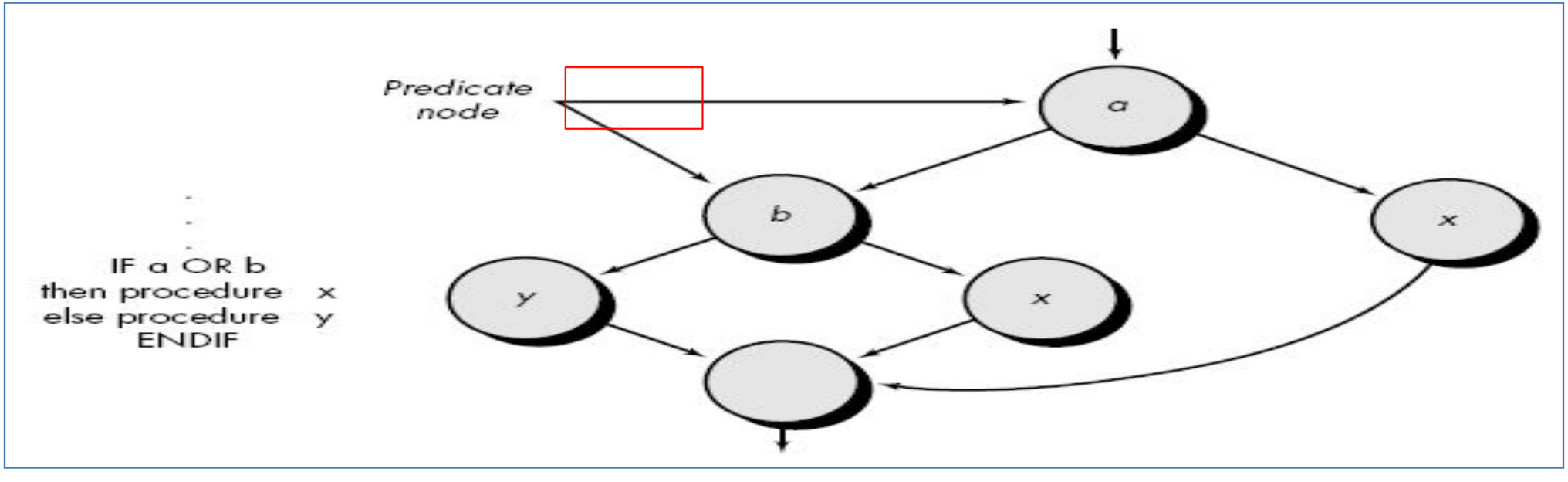
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing

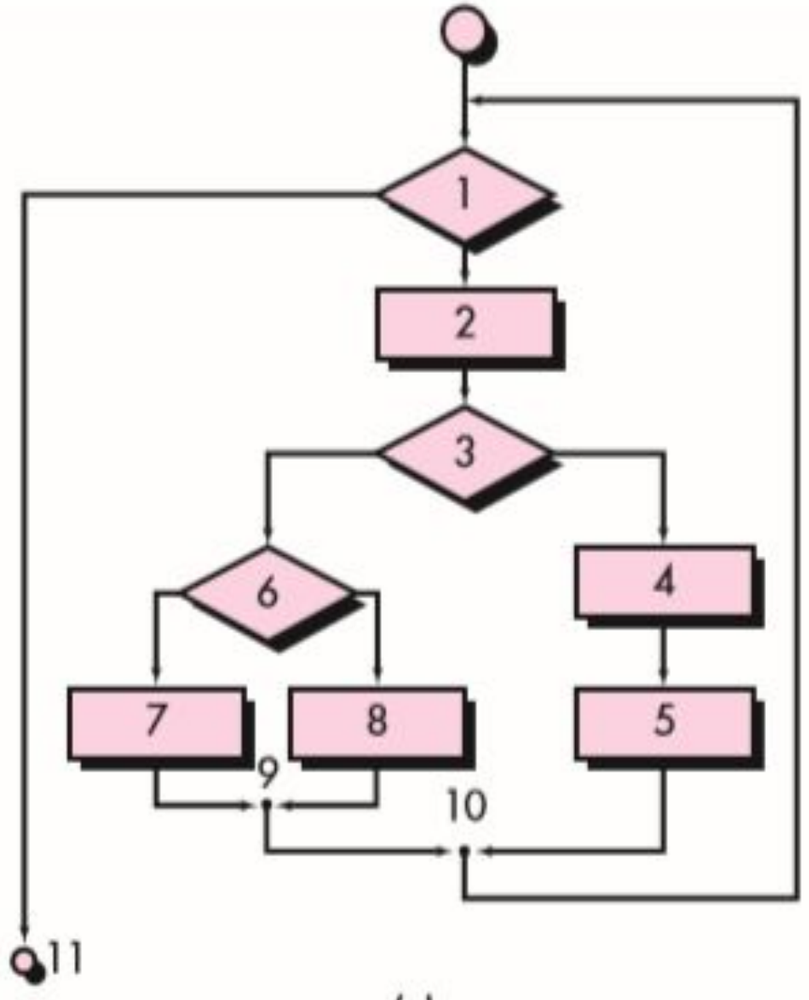
# Flow Graph Notation

- A circle in a graph represents a **node**, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a **predicate node**
  - A predicate node has two edges leading out from it (True & False)
- An **edge**, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called **regions**
- When counting regions, include the area outside the graph as a region, too

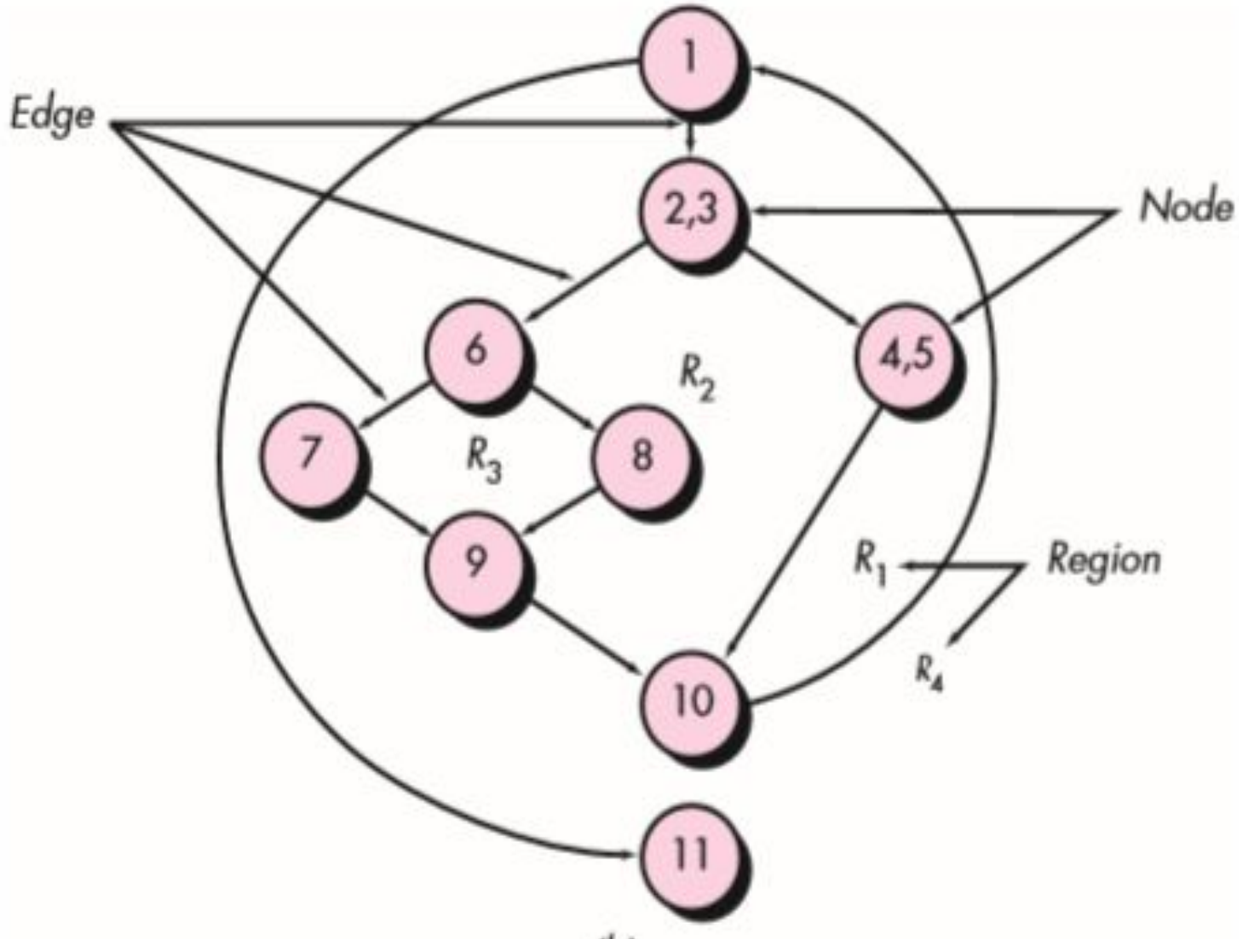
# Basis Path Testing: Flow Graph Notation



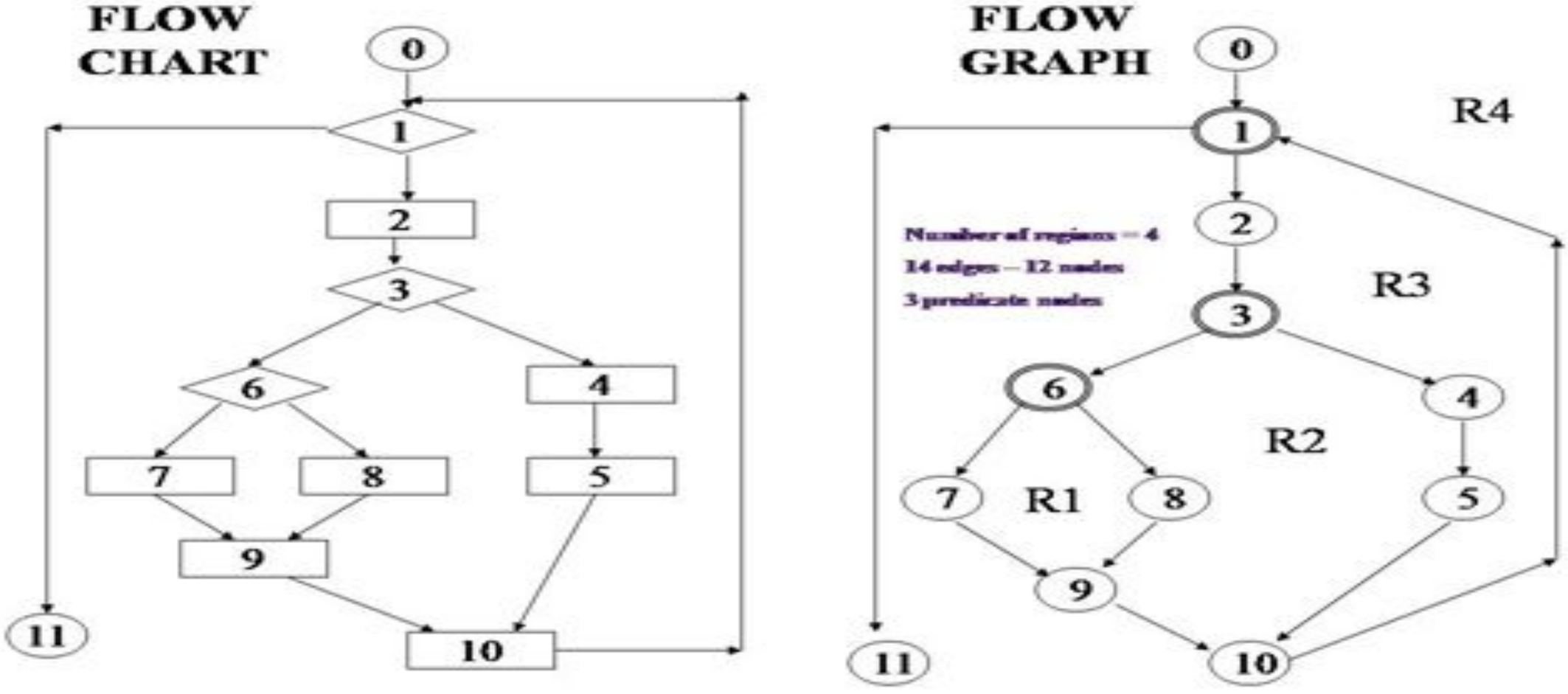



(a)




(b)

# Flow Graph Example



# Independent Program Paths

- Basis set for flow graph on previous slide
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity

# Cyclomatic Complexity

- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
  - **The number of regions**
  - **V(G) = E − N + 2**, where E is the number of edges and N is the number of nodes in graph G
  - **V(G) = P + 1**, where P is the number of predicate nodes in the flow graph G

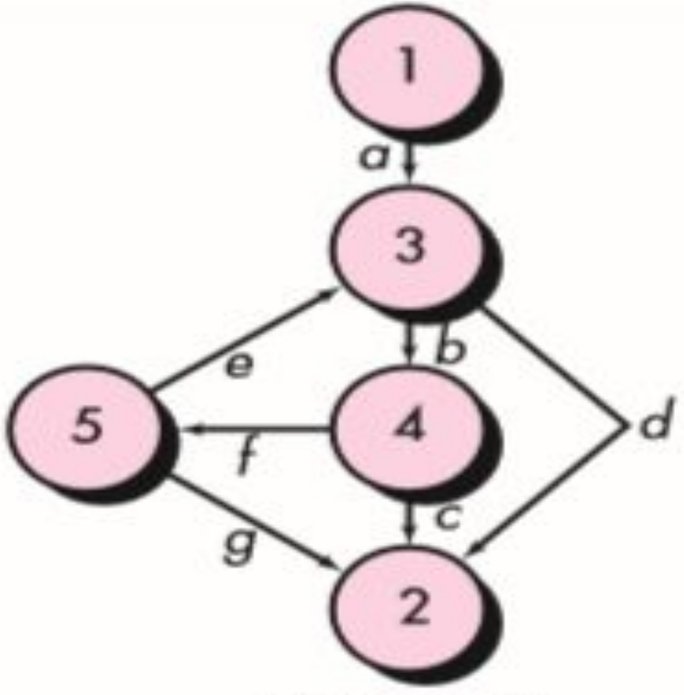
- Results in the following equations for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges − 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

# Deriving Test Cases

- *The following steps can be applied to derive the basis set:*
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

# Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing




Flow graph

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

Graph matrix

# Black-box Testing

- Complements white-box testing by uncovering different classes of errors
- Focuses on the functional requirements
- Used during the later stages of testing after white box testing
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program

## *Black-box Testing Categories*

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external data base access
- Behavior or performance errors
- Initialization and termination errors

| Black Box Testing | White Box Testing |
| --- | --- |
| 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. |
| This type of testing is carried out by testers. | Generally, this type of testing is carried out by software developers. |
| Implementation Knowledge is not required to carry out Black Box Testing. | Implementation Knowledge is required to carry out White Box Testing. |
| Programming Knowledge is not required to carry out Black Box Testing. | Programming Knowledge is required to carry out White Box Testing. |
| 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 |
| Black box testing means functional test or external testing. | White box testing means structural test or interior testing. |
| In Black Box testing is primarily concentrate on the functionality of the system under test. | In White Box testing is primarily concentrate on the testing of program code of the system under test like code structure, branches, conditions, loops etc. |
| The main aim of this testing to check on what functionality is performing by the system under test. | The main aim of White Box testing to check on how System is performing. |
| Black Box testing can be started based on Requirement Specifications documents. | White Box testing can be started based on Detail Design documents. |
| The Functional testing, Behavior testing, Close box testing is carried out under Black Box testing, so there is no required of the programming knowledge. | The Structural testing, Logic testing, Path testing, Loop testing, Code coverage testing, Open box testing is carried out under White Box testing, so there is compulsory to know about programming knowledge. |