

소프트웨어공학 원리

(SEP521)



Software Testing – II

Jongmoon Baik



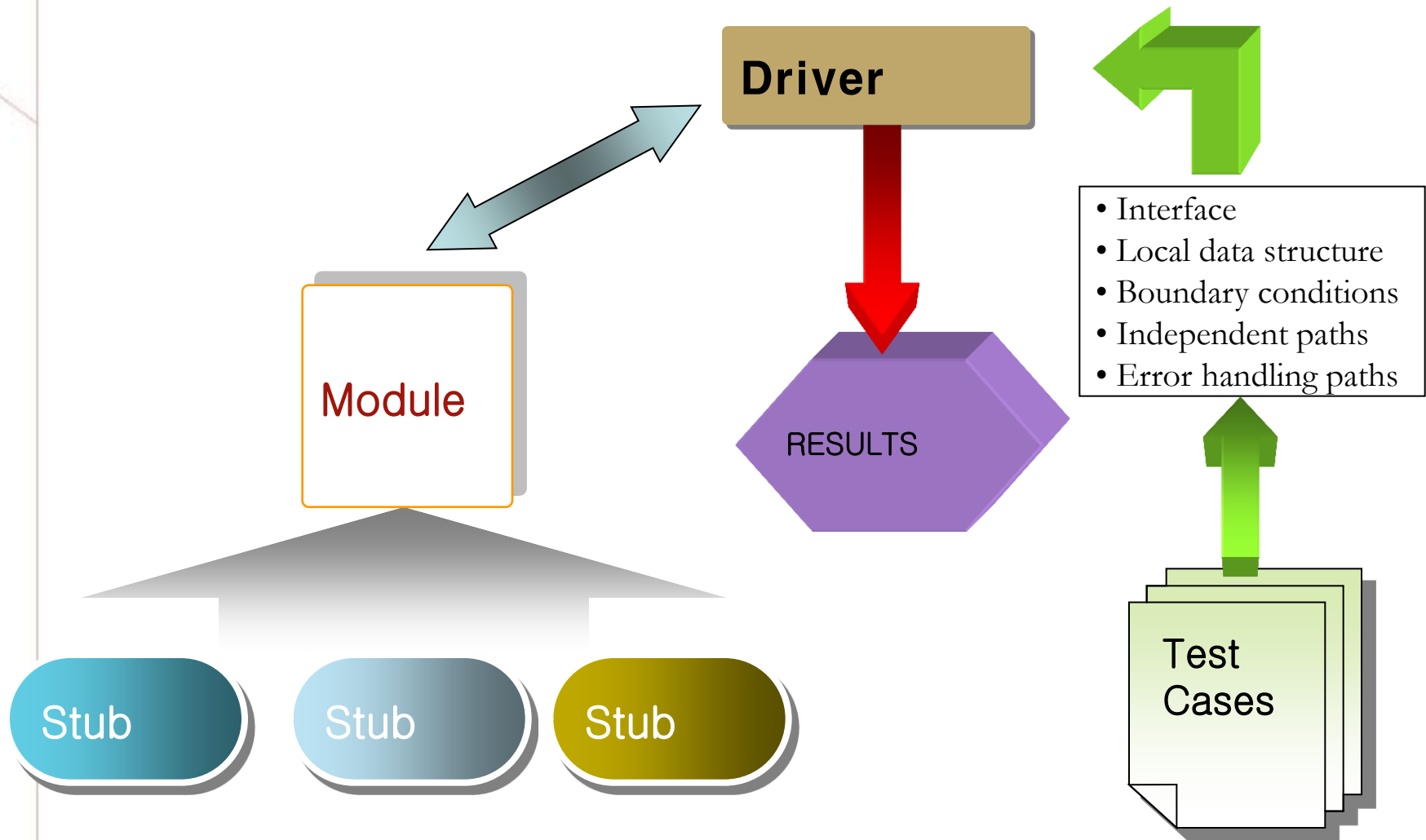
Unit (Component) Testing

- Focus on the smallest software design (module or component)
- Often corresponds to the notion of “compilation unit” from the prog. Language
- Responsibility: Developer
- Test internal processing logic and data structure within the boundary of a component
- Can be conducted in parallel for multiple components
- May be necessary to create stubs: “fake” code that replaces called modules
 - If not yet implemented, or not yet tested

What are tested in Unit Testing?

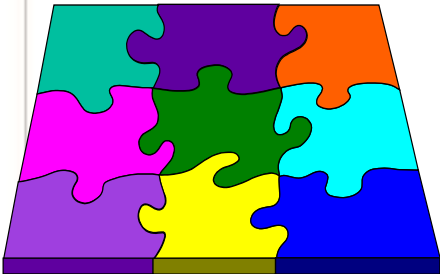
- Information flows for module interfaces
- Local data structures
- All independent paths (basis path) through control structure
- Boundary condition
- Error handling paths

Unit Test Environment



Integration Testing

- Exercise two or more combined units (or components)
- Main objectives:
 - Detect interface errors
 - Assure the functionalities when combined
- Responsibility: Developers or Testing Group



■ Issues

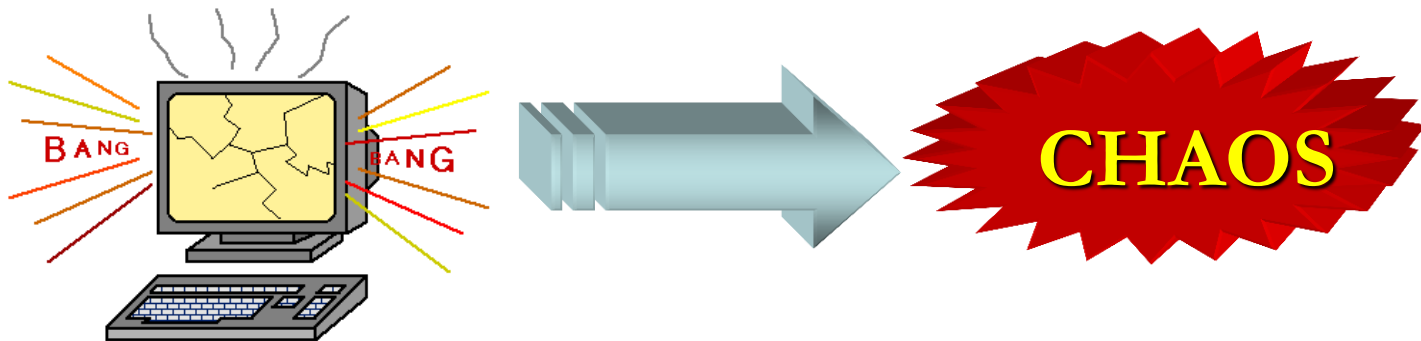
- Integration Strategy (How to Combine?)
- Integration with thirty-party components
 - Compatibility, Correctness, etc

Integration Testing Strategies

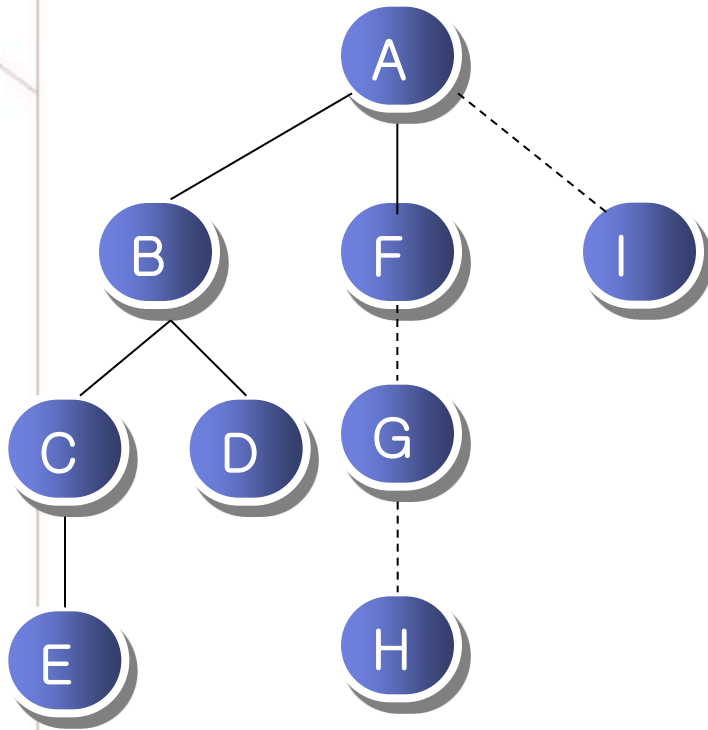
- Non- Incremental Integration
 - “Big Bang” approach
- Incremental Approaches
 - Top-down Integration
 - Bottom-up Integration
 - Sandwich Testing

"Big Bang" Approach

- All unit tested components are combined at once and tested as whole
- Disadvantages
 - Difficult to correct defects
 - Critical and peripheral modules not distinguished
 - When errors are corrected, new ones appear : endless loop
 - User does not see the product until very late in the development life cycle



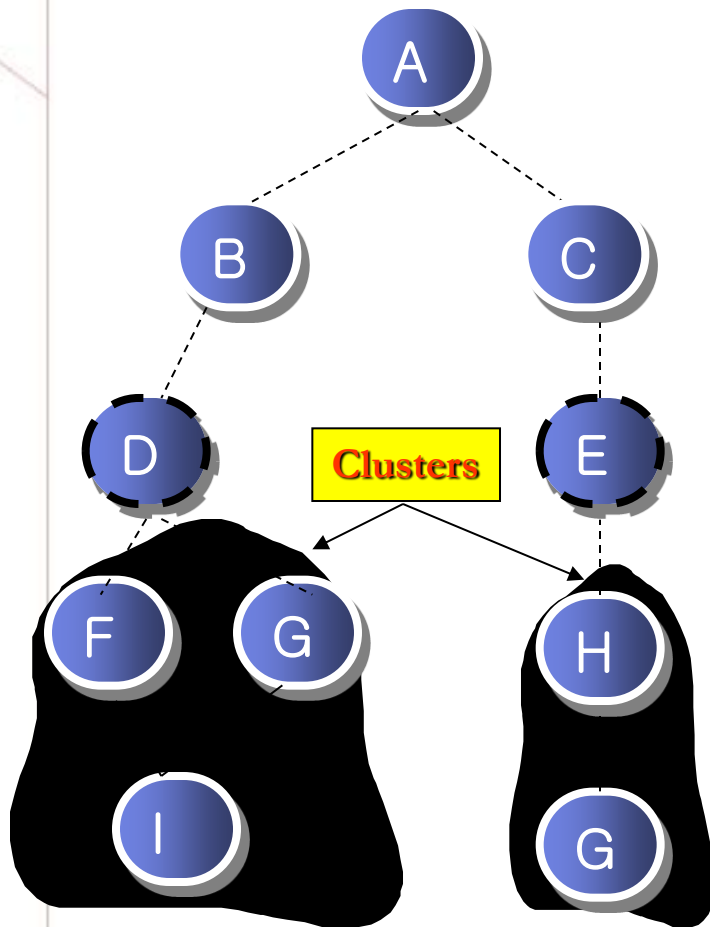
Top-down Integration



Depth-First Approach

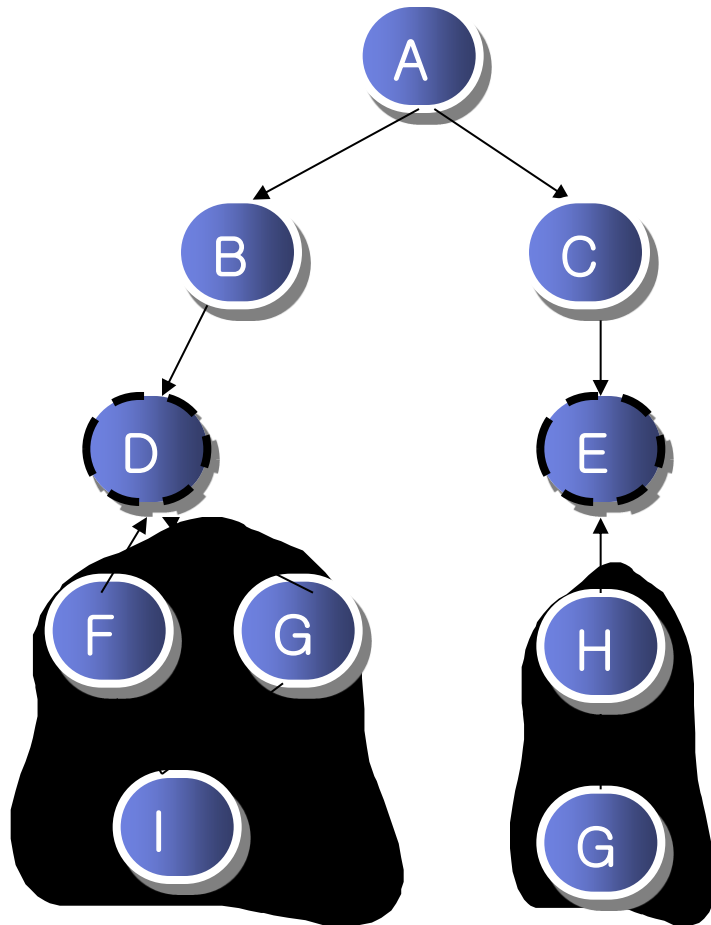
- An Incremental Approach to construction of the software architecture
- Integrated by moving downward through the control hierarchy
 - Depth-First or Breadth-First
- Begins with main control module (main program)

Bottom-up Integration



- Begins construction and testing with atomic modules
 - From components at the lowest levels in the program structure
- No need for stubs
- Drivers are replaced one at a time

Sandwich Integration



- Top level modules are tested with stubs
- Worker modules are integrated into clusters
- Advantages:
 - Significantly Reduced number of drivers
 - Simplified integration of clusters

Regression Testing

- Re-execution of some subset of tests to see if anything is broken by a change
 - Can be applied to unit, integration, and system testing
- Require automatic test suit to be practical
 - Impractical and inefficient to re-execute every test for every program function once a change has occurred
 - Prioritization of test cases (maximize defect detection rate)
- Regression testing is an integral part of the XP software development methodology

Smoke Testing

- Designed as a pacing mechanism for time-critical projects
 - Assess its project on a frequent basis
- Analogy to testing electrical circuits:
 - plug it in and see if it smokes.
- Main objective:
 - to detect “Show Stopper”
- Benefits:
 - Minimized integration risks
 - Improved quality of the end product
 - Simplified error diagnosis and correction
 - Easier progress to access

How to select a strategic option?

- Depends upon software characteristics and project schedule.
- Identify critical modules and test them as early as possible
 - Critical Module's characteristics:
 - Address several software requirements
 - Has a high level of control
 - Complex and error-prone
 - Has definite performance requirements
- Focus on critical module functions in regression tests

Validation Testing

- Intended to show that the software meets its requirements.
 - Focus on user-visible actions and user-recognizable output from the system
- A successful test is one that shows that a requirements has been properly implemented. (Conformity)
- A deviation or error uncovered at this stage can be rarely corrected prior to scheduled delivery
 - Necessary to negotiate with customer to establish a method for resolving deficiencies

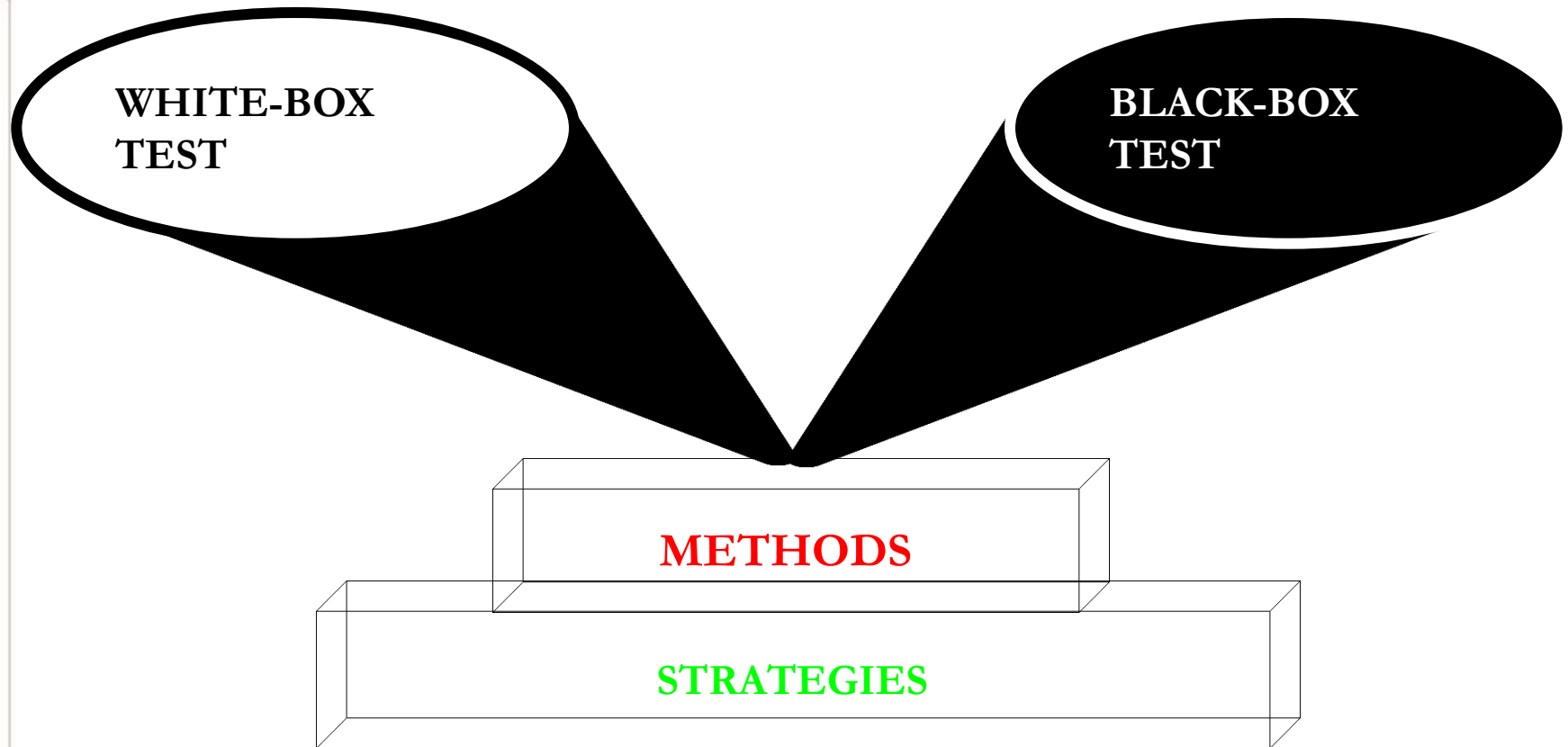
Alpha test vs. Beta test

- Alpha test
 - Conducted at developers' site by end-users
 - Under a controlled environment
 - is often employed for off-the-shelf software as a form of internal acceptance testing
- Beta test
 - Conducted at end-user sites
 - “live application” of the software in a non-controlled environment
 - available to the open public to increase the feedback field to a maximal number of future users

System Testing

- Test the system's compliance with its specified requirements as a whole.
 - After software is incorporated with other system elements (e.g.: Hardware, People, Information)
 - A series of different tests to fully exercise the computer-based system
- Types of system tests
 - Recovery Testing
 - Security Testing
 - Stress Testing
 - Performance Testing

Software Testing



Black-box test vs. White-box test

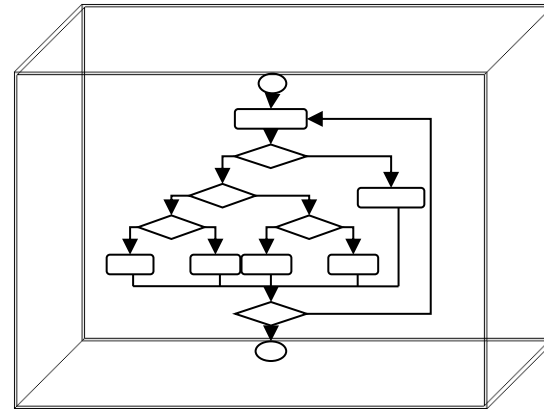
Black-box test

- *Functional or behavioral testing*
- Conducted at software interface
- Examines some fundamental aspect of a system
- Ignores internal logic of a software system

White-box test

- *Glass-box or structural testing*
- Uses knowledge of the internal structure of the software
- Examine procedural detail (logical paths and collaboration b/w components)

White-box Test



Basis Path Testing

- A white-box testing technique
- Enable to derive a logical complexity measure of a procedural design
- Provide a guideline defining a basis set of execution paths
- Guarantee to execute every statement in the program at once

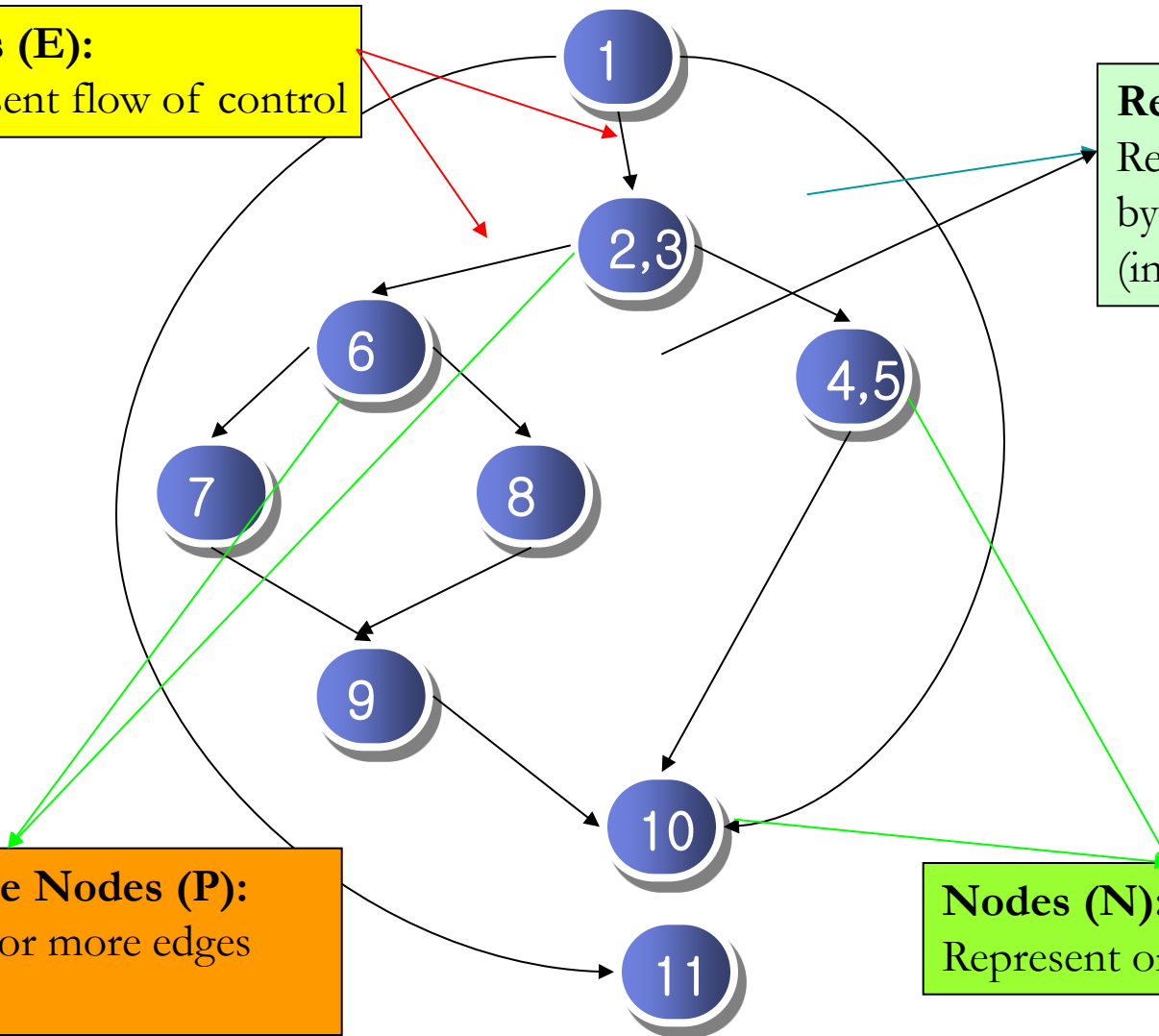
Flow Graph

Edges (E):

Represent flow of control

Regions (R):

Represent areas bounded by edges and nodes
(includes outside of graph)



Predicate Nodes (P):

Has two or more edges from it

Nodes (N):

Represent one or more statements

Independent Paths?

- Any path through the program that introduces at least one new set of processing statements or a new condition
- Test can be designed to force execution of these paths (a basis set)
- Guaranteed to execute every statement at least once

Steps to Derive Test Cases

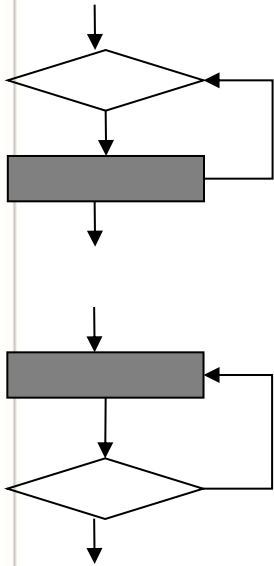
1. Using the design or code as a foundation, draw a corresponding flow graph
2. Determine the cyclomatic complexity of the flow graph
3. Determine a basis set of linearly independent paths
4. Prepare test cases that will force execution of each path in the basis set

Other Control Structure

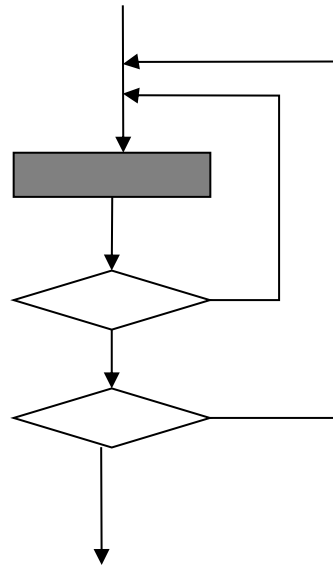
- Condition Testing:
 - Focus on exercising the logical condition
 - Simple and compound condition (s) : Boolean variables or relational expression
- Data Flow Testing:
 - Select test paths of a program according to the location s of definitions and uses of variables
 - Define-Use (DU) testing strategy
 - To require that every DU chain be covered at least once
 - No guarantee the coverage of all branches of a program

Loop Testing

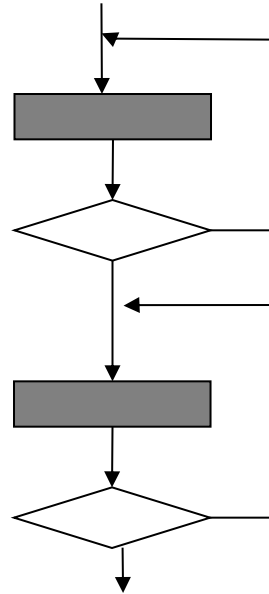
- Focus exclusively on the validity of loop conditions



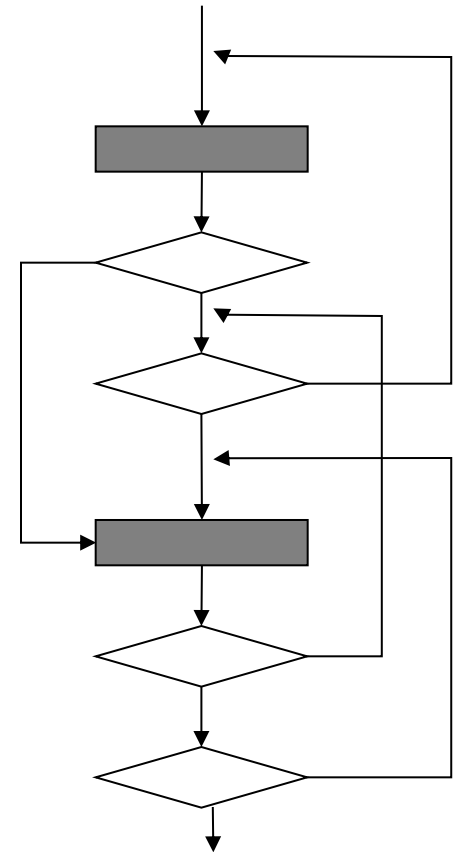
Simple Loop



Nested Loop



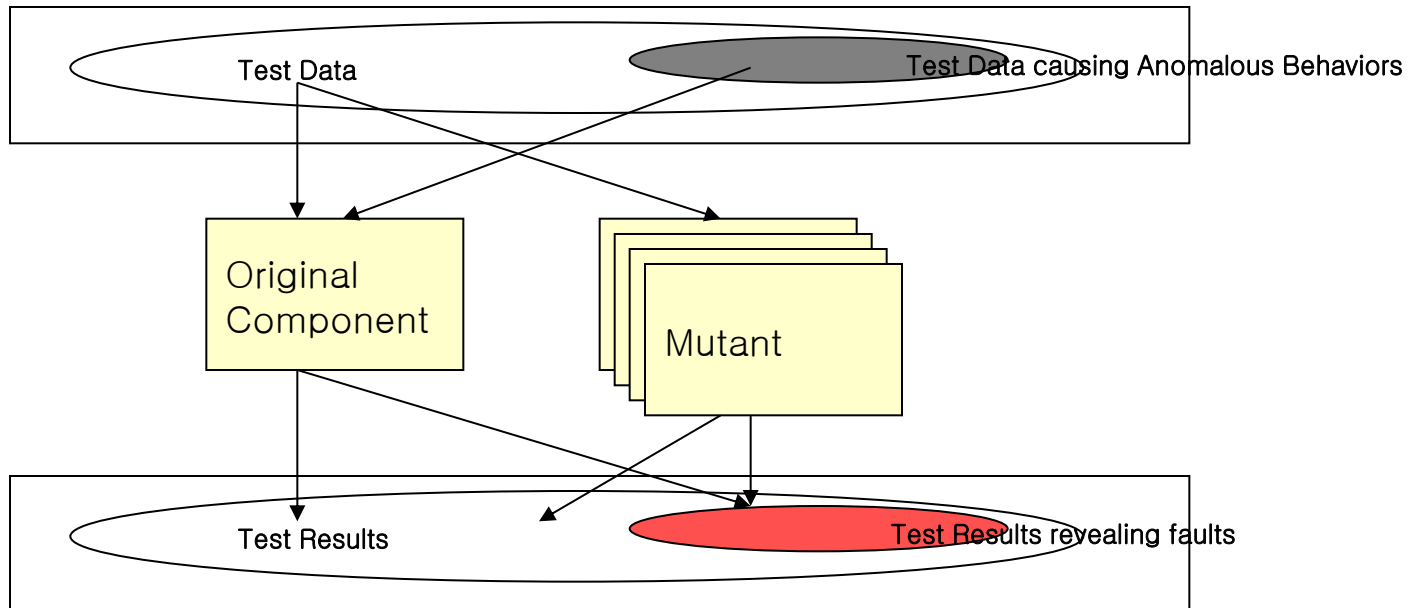
Concatenated Loop



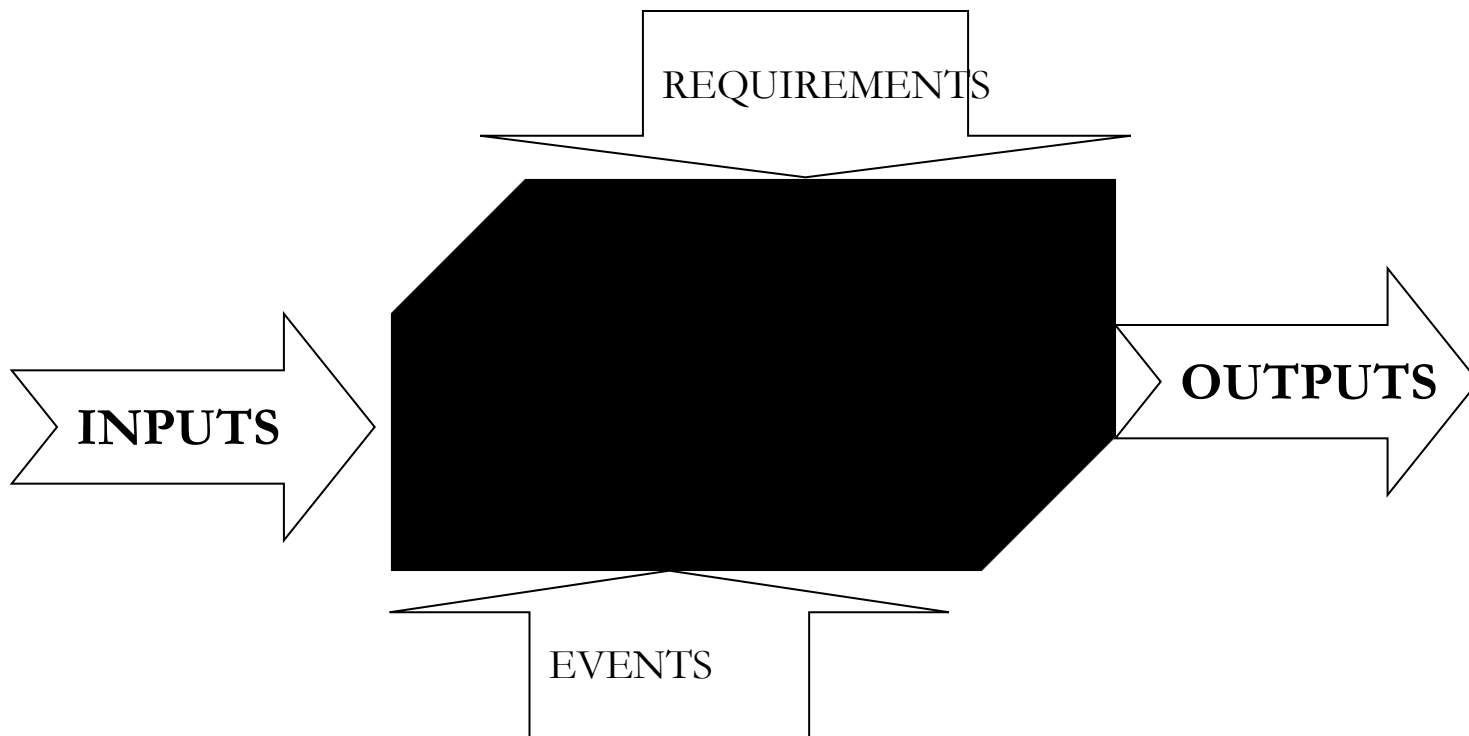
Unstructured Loop

Mutation Testing (Fault Injection)

- Test run with mutants which are similar components modified from the original components
 - If the test data reveals the fault in the mutant, kill the mutant
 - If not, the tests can not distinguish the original from the mutant
 - Develop additional test data to reveal the fault and kill the mutant



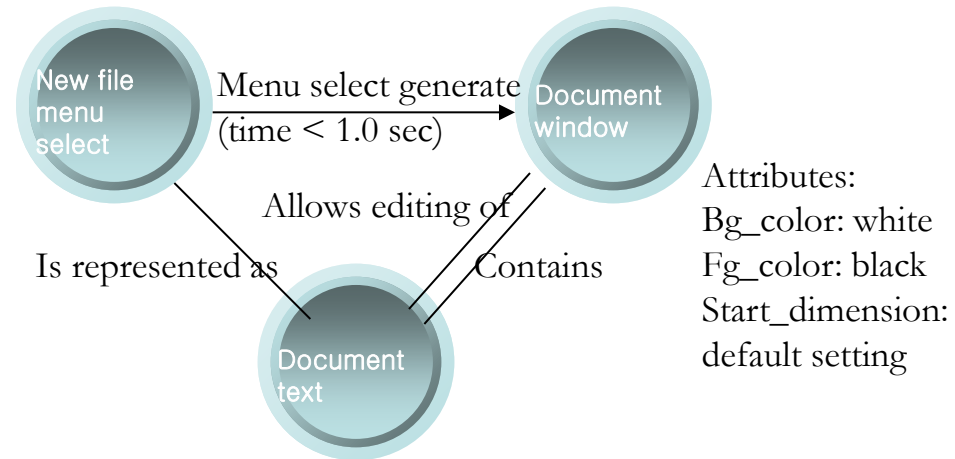
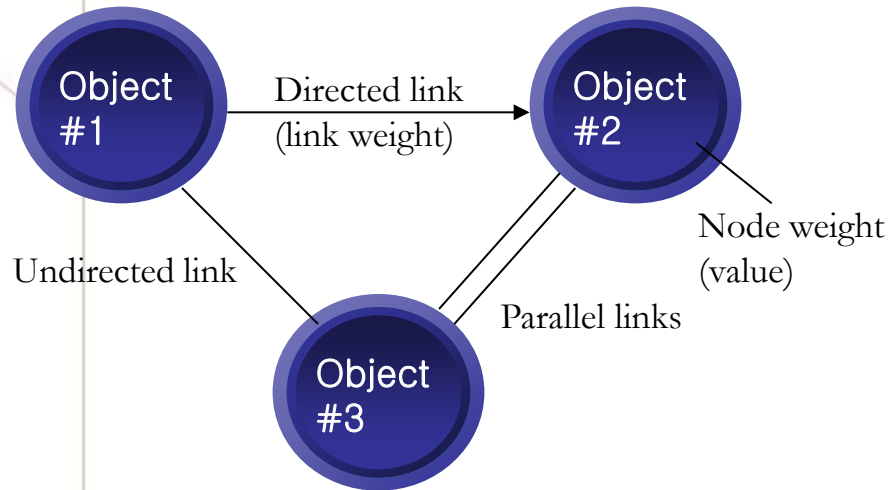
Black-box Testing



How to Design Black-box tests?

- Questions to be answered for designing tests
 - How is functional validity tested?
 - How is system behavior and performance tested?
 - What classes of input will make good test cases?
 - Is the system particularly sensitive to certain input values?
 - How are the boundaries of a data class isolated?
 - What data rates and data volume can the system tolerate?
 - What effect will specific combinations of data have on system operation?

Graph-Based Testing Method



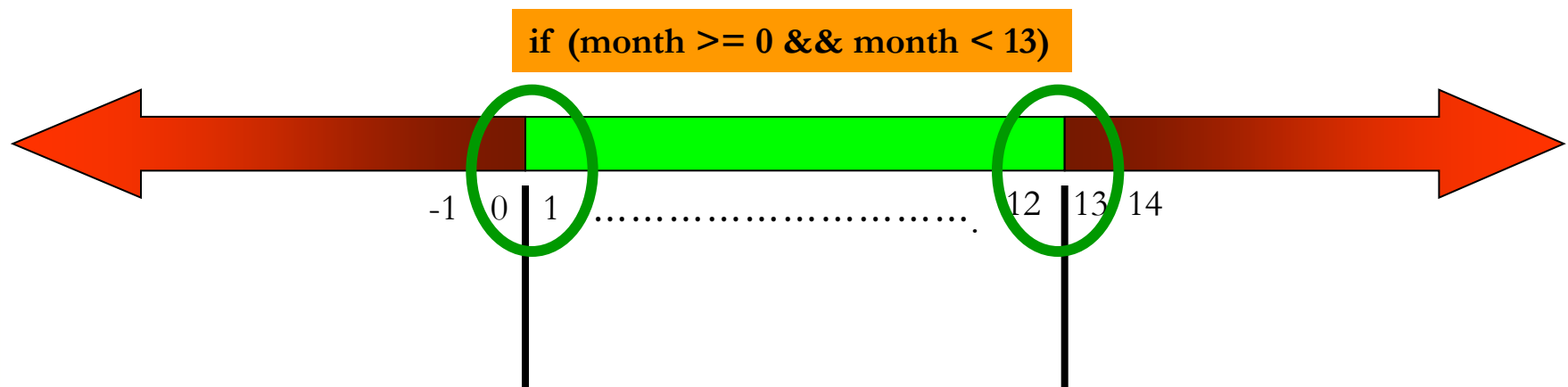
- First Step in Black-box testing
 - Understand the objects modeled in software and relationships among these objects
- Verify all objects have the expected relationship to one another
 - Design test cases by traversing the graph and covering all the relationships to uncover errors

Equivalence Partitioning

- Divides the input domain of a program into classes of data
- Based on an evaluation of equivalent classes for an input condition.
 - An equivalent class: a set of valid or invalid states for input conditions
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- A guideline (not a rule) to define equivalence classes
 - If an input condition specifies a range, define one valid and two invalid equivalence classes
 - If an input condition is Boolean, define one valid and one invalid equivalence classes
 - If an input condition requires specific value, define one valid and two invalid equivalence classes
 - If an input condition specifies a member of set, define one valid and one invalid equivalence classes

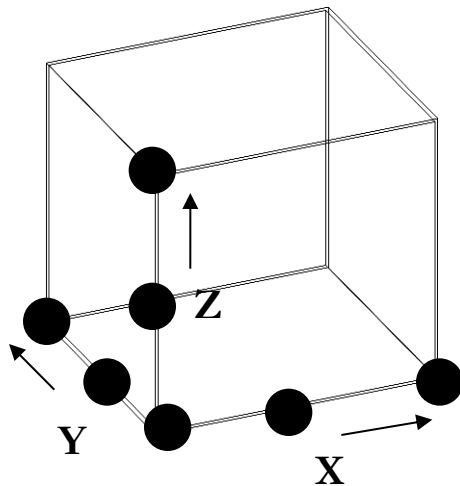
Boundary Value Analysis

- Focus on selecting a set of test cases that exercise bounding values
 - Select test cases at the edges of the class
- Complementary to Equivalence Partitioning
- Derive test cases from output domain as well

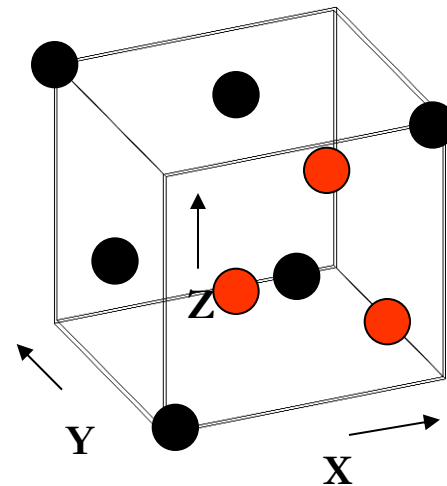


Orthogonal Array Testing

- Can be used when the number of input parameters and their values are clearly bounded
- Support more complete test coverage



One input items at a time



L9 orthogonal array

Q & A

