Software Testing Techniques

- Testing fundamentals

- White-box testing

- Black-box testing

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*.)

Software Testing

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

Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.

- A good test case is one that has a high probability of finding an as-yet-undiscovered error.

- A successful test is one that uncovers an as-yet-undiscovered error.

What Testing Shows

errors

requirements conformance

performance

an indication of quality
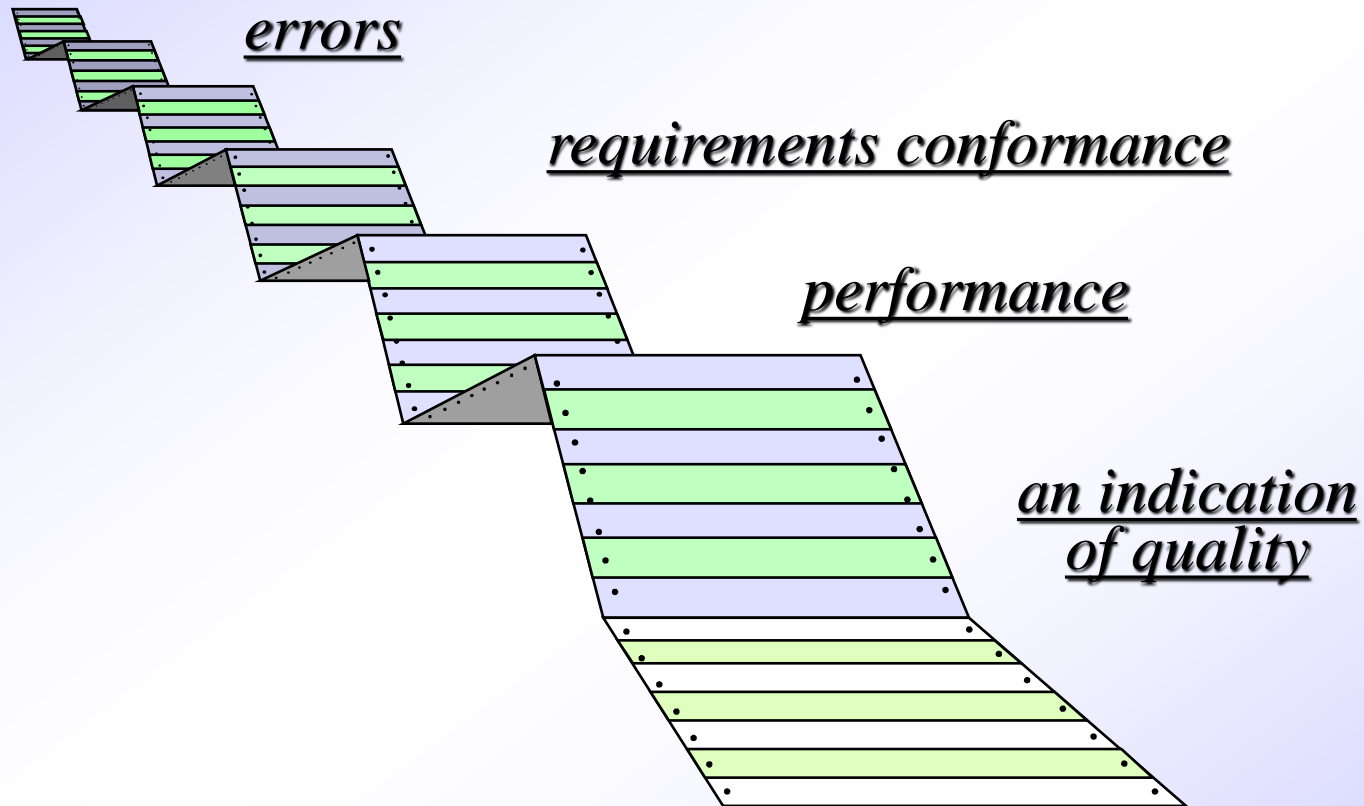
Who Tests the Software?

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_**

Characteristics of Testable Software

- Operable
  - The better it works (i.e., better quality), the easier it is to test
- Observable
  - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
  - The states and variables of the software can be controlled directly by the tester
- Decomposable
  - The software is built from independent modules that can be tested independently

(more on next slide)

Characteristics of Testable Software (continued)

- Simple
  - The program should exhibit functional, structural, and code simplicity
- Stable
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
  - The architectural design is well understood; documentation is available and organized

GOOD 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; one test should not serve the same purpose as another test
- 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
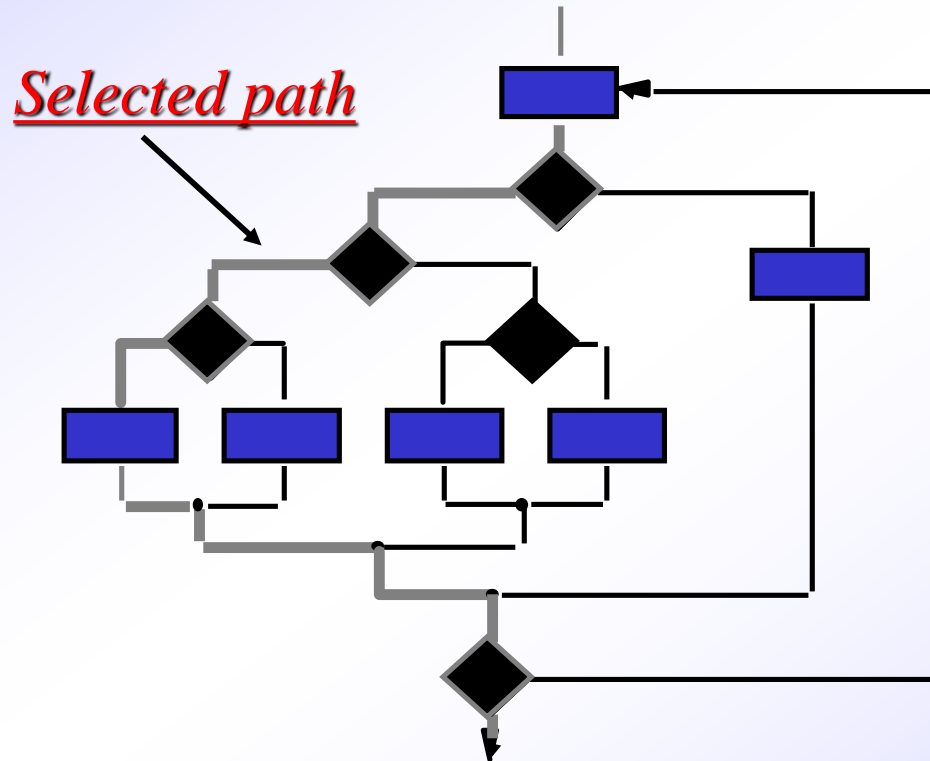
(sompurno) Exhaustive Testing
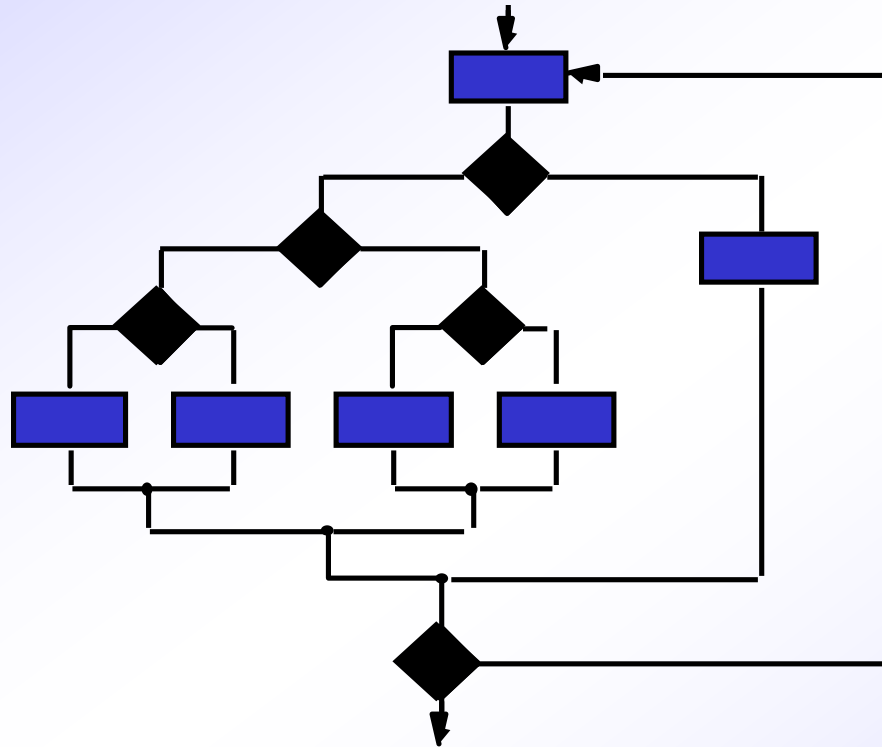
loop < 20 X

$\dfrac{10^{14}}{1000*60*60*24*365}$

There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!
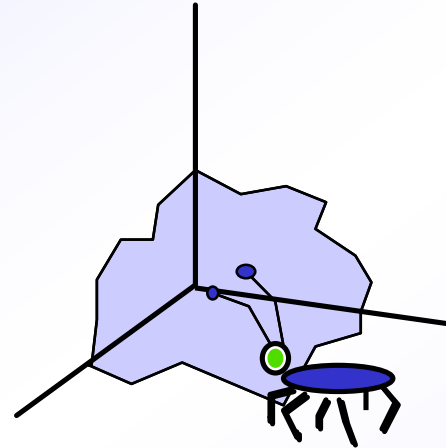
Selective Testing

Selected path

loop < 20 X

Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."
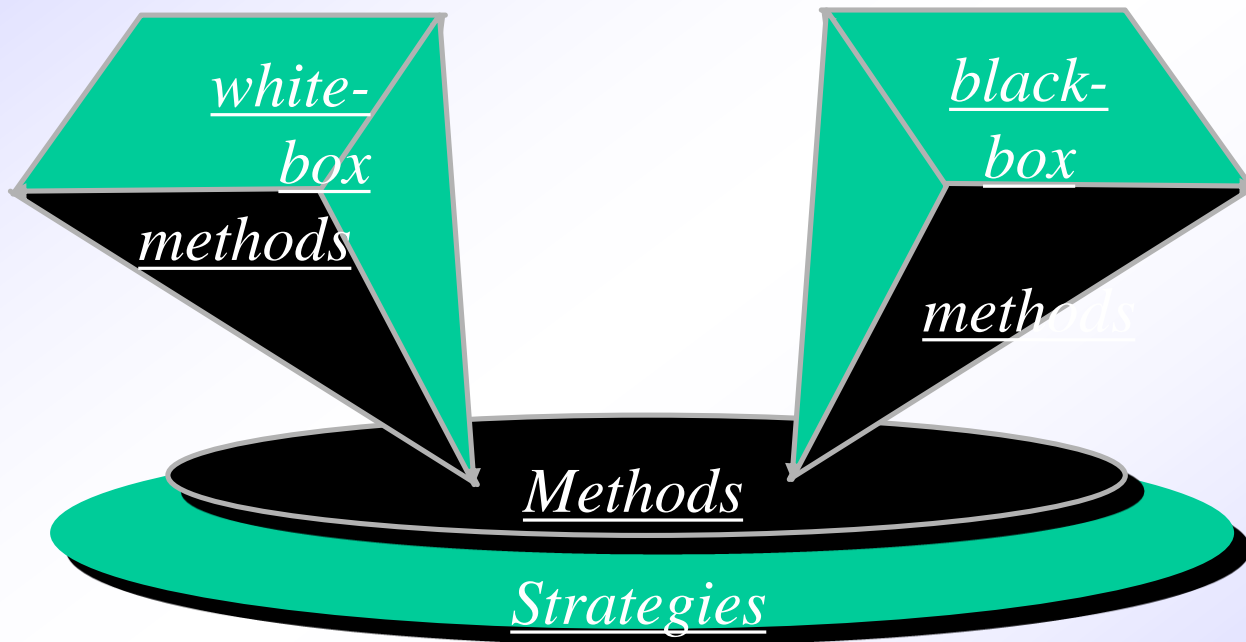
Boris Beizer

OBJECTIVE   *to uncover errors*

CRITERIA   *in a complete manner*

CONSTRAINT   *with a minimum of effort and time*

Software Testing

white-box methods

*black-box methods*

Methods

Strategies

Two Unit Testing Techniques

- Black-box testing
  - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software
- White-box testing
  - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
  - Involves tests that concentrate on close examination of procedural detail
  - Logical paths through the software are tested
  - Test cases exercise specific sets of conditions and loops

White-box Testing

White-Box Testing
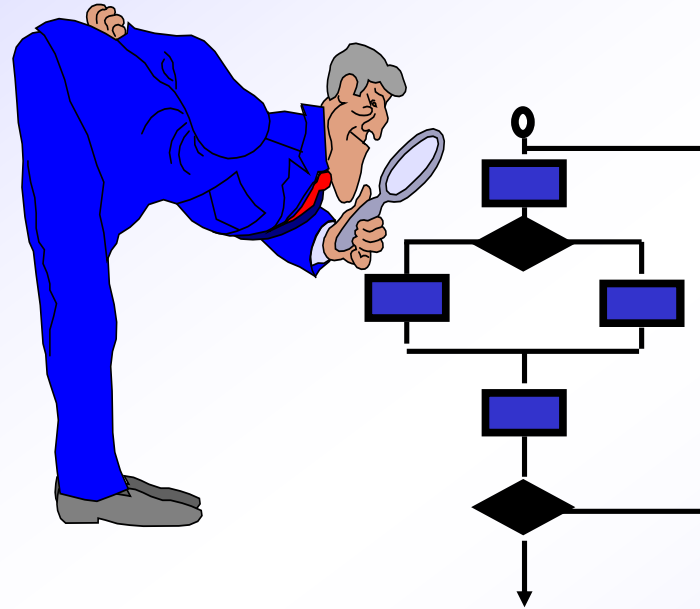
... our goal is to ensure that all statements and conditions have been executed at least once ...

White-box Testing

- Uses the control structure part of component-level design to derive the test cases

- These test cases   (not all possible path)
    - 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

- White-box testing technique proposed by Tom McCabe

- 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 <u>node</u>, which stands for a <u>sequence</u> of one or more procedural statements
- A node containing a simple conditional expression is referred to as a <u>predicate node</u>
  - Each <u>compound condition</u> in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has <u>two</u> edges leading out from it (True and False)
- An <u>edge</u>, 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 <u>regions</u>
- When counting regions, include the area outside the graph as a region, too
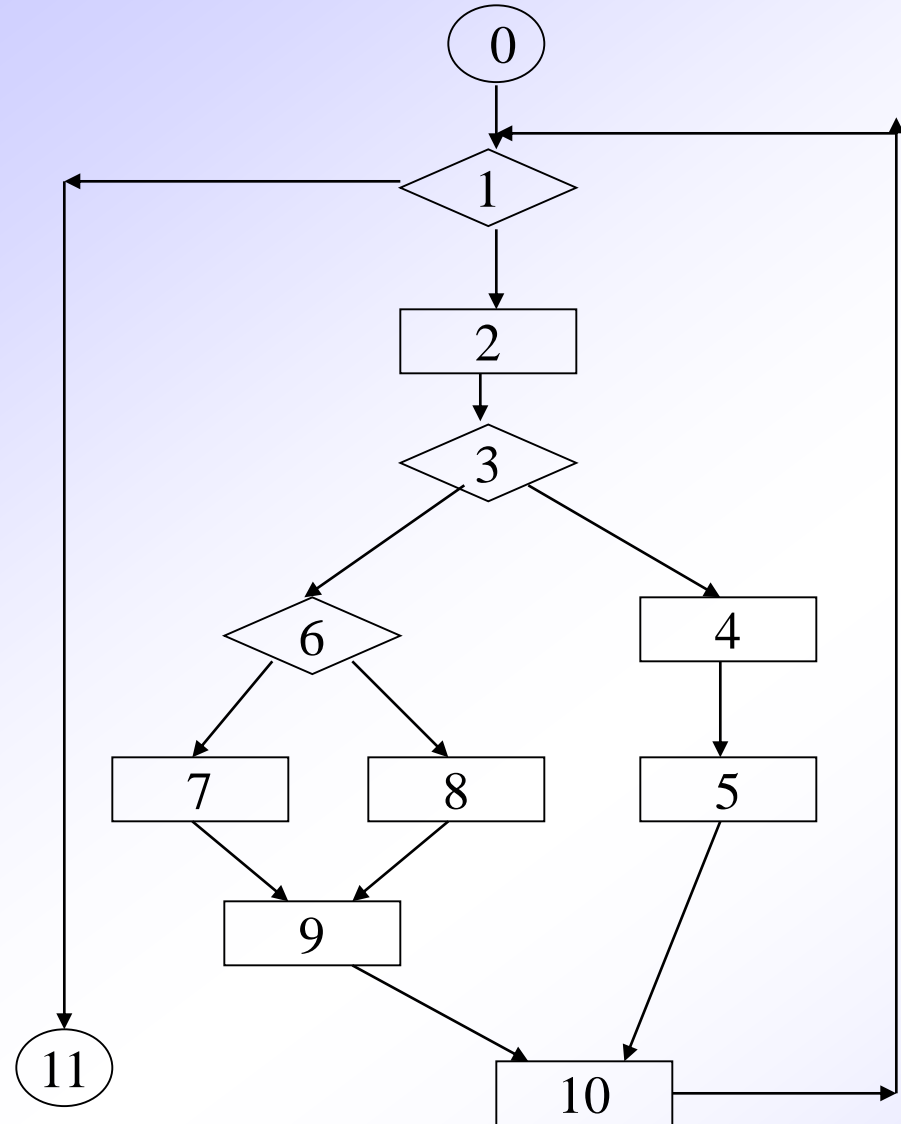
Flow Graph Notation

The structured constructs in flow graph form:

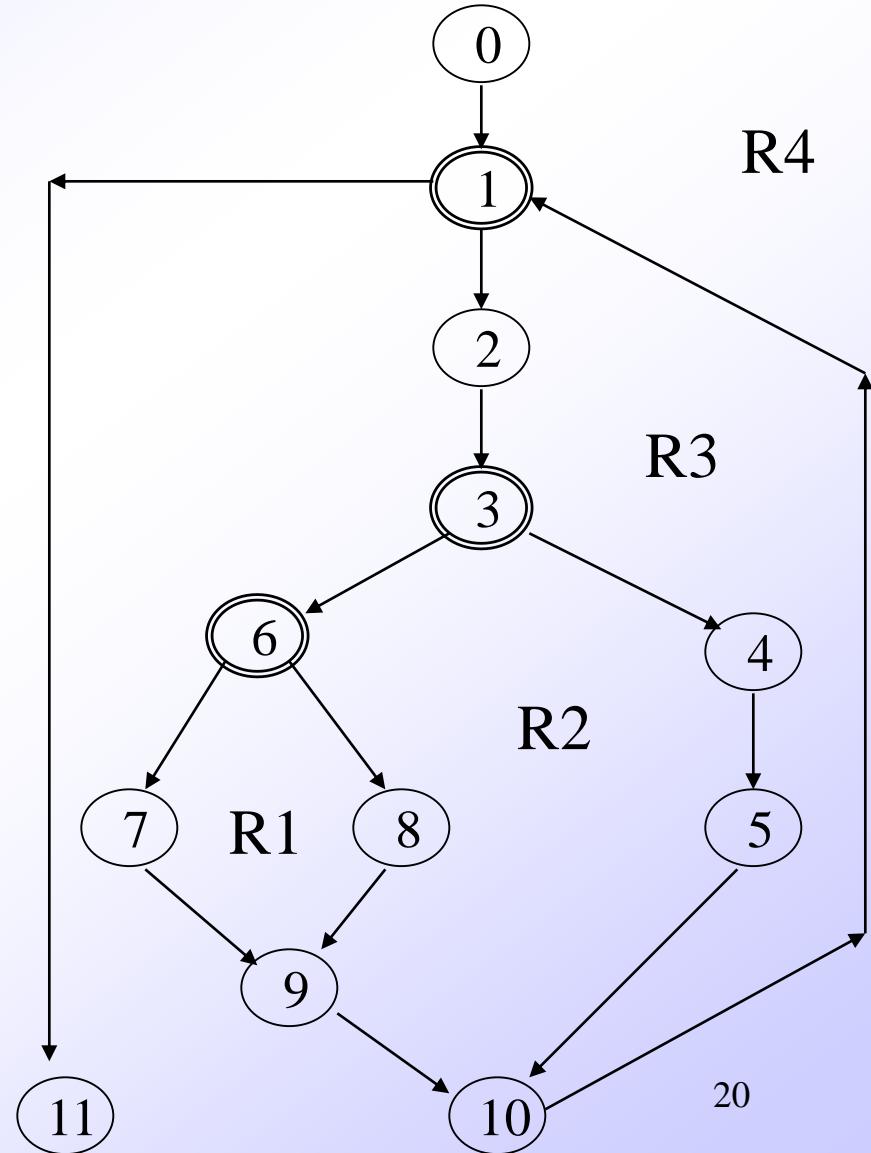Sequence   If   While   Until   Case
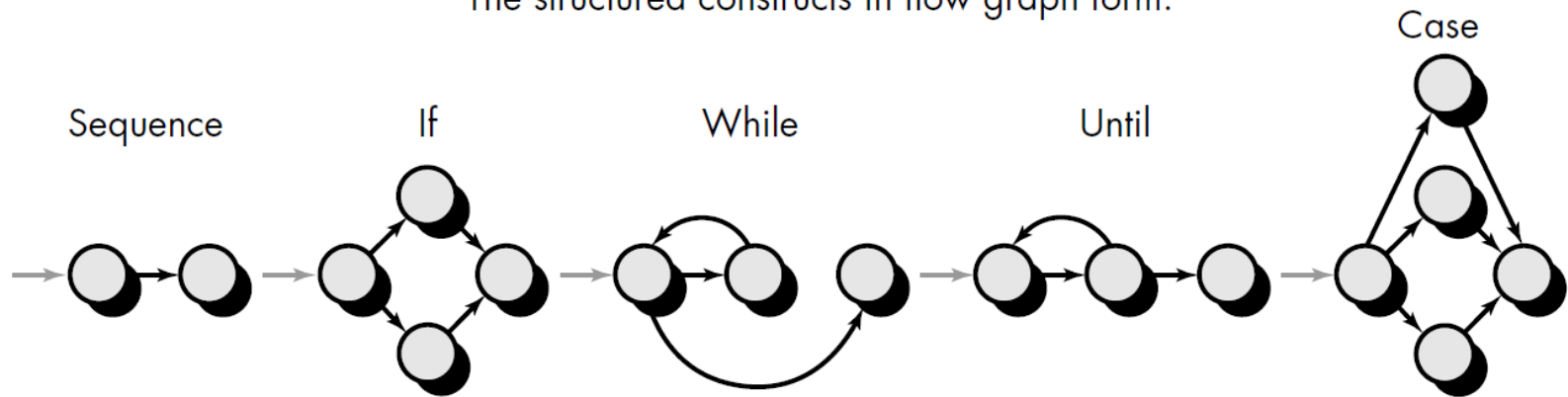
Where each circle represents one or more
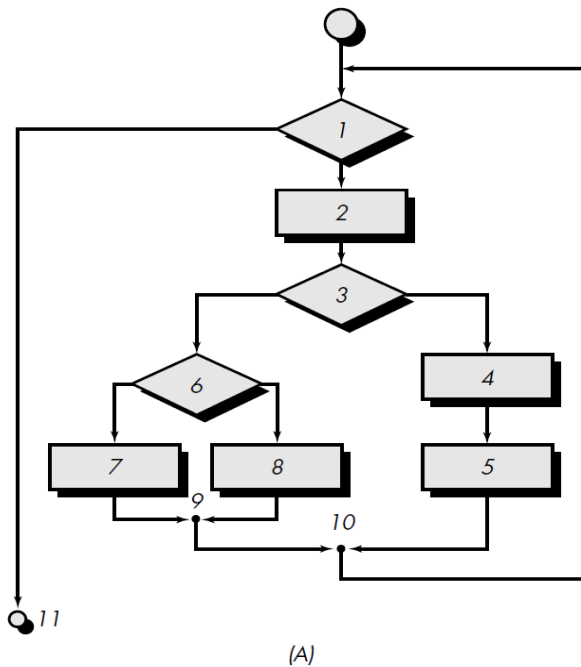nonbranching PDL or source code statements

Flow Graph Example

FLOW CHART

FLOW GRAPH
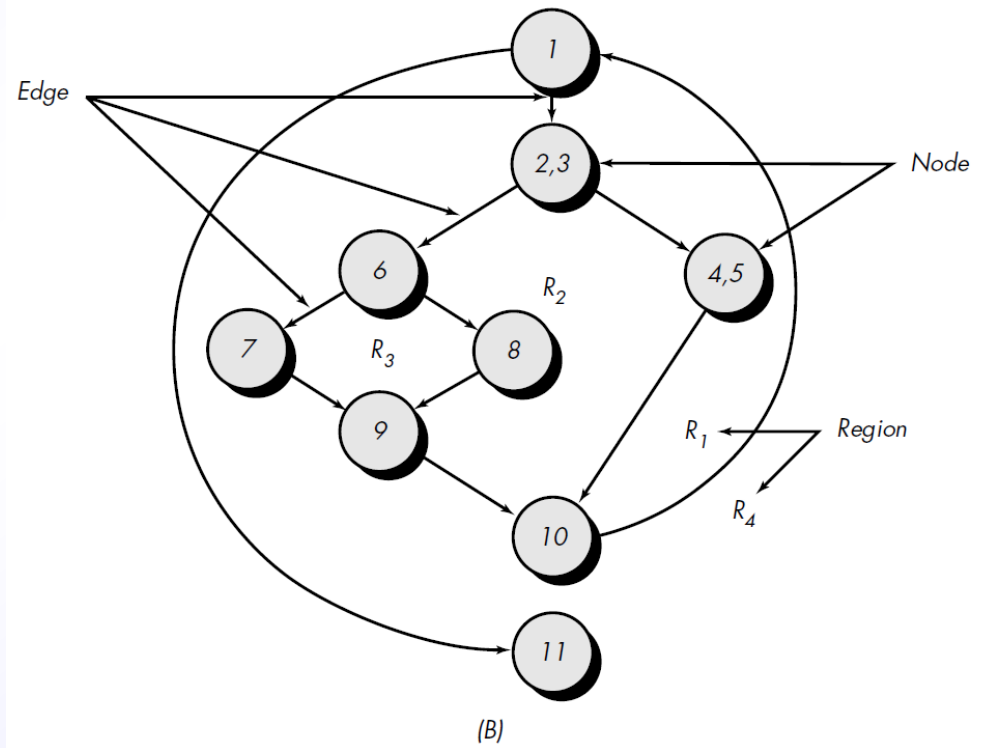
Flow Graph Example

FLOW CHART

FLOW GRAPH

(A)

(B)
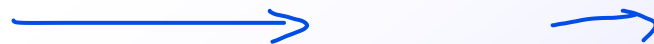
Independent Program Paths

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)

- Must move along <u>at least one</u> edge that has not been traversed before by a previous path

- Basis set for flow graph on previous example

  input set   outputset

  - Path 1: 0-1-11   test case: $i > n$ → 101, 102 ····· ··
  - 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

- Note that each new path introduces a new edge. The path

  - 0-1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

- The <u>number of paths</u> in the basis set is determined by the <u>cyclomatic</u> complexity

Cyclomatic Complexity

- Provides a quantitative measure of the <u>logical complexity</u> of a program
- Defines the <u>number of independent paths</u> in the basis set
- Provides an <u>upper bound</u> for the number of tests that must be conducted to ensure <u>all statements</u> have been executed <u>at least once</u>
- Can be computed <u>three</u> 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 the Basis Set and Test Cases

1) Using the design or code as a foundation, draw a corresponding flow graph

2) Determine the cyclomatic complexity of the resultant 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

```
prime number generation between 1 to n

public static void main(String[] args){
  int n=100;
  out:
  for(int i=2;i<=n;i++){
   for(int j=(int) Math.sqrt(i);j>=2;j--){
    if(i%j==0){
     continue out;
    }
   }
   System.out.println(i);
  }
}
```

Graph Matrices

FIGURE 17.6
Graph matrix

Flow graph
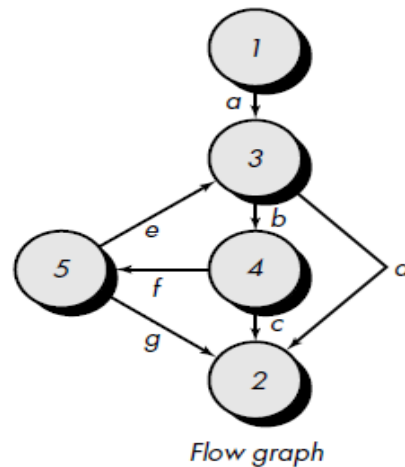
Graph matrix

FIGURE 17.7
Connection matrix

Graph matrix

25

A Second Flow Graph Example

```
1   int functionY(void)
2   {
3       int x = 0;
4       int y = 19;

5   A: x++;
6       if (x > 999)
7           goto D;
8       if (x % 11 == 0)
9           goto B;
10      else goto A;

11  B: if (x % y == 0)
12          goto C;
13      else goto A;

14  C: printf("%d\n", x);
15      goto A;

16  D: printf("End of list\n");
17      return 0;
18  }
```
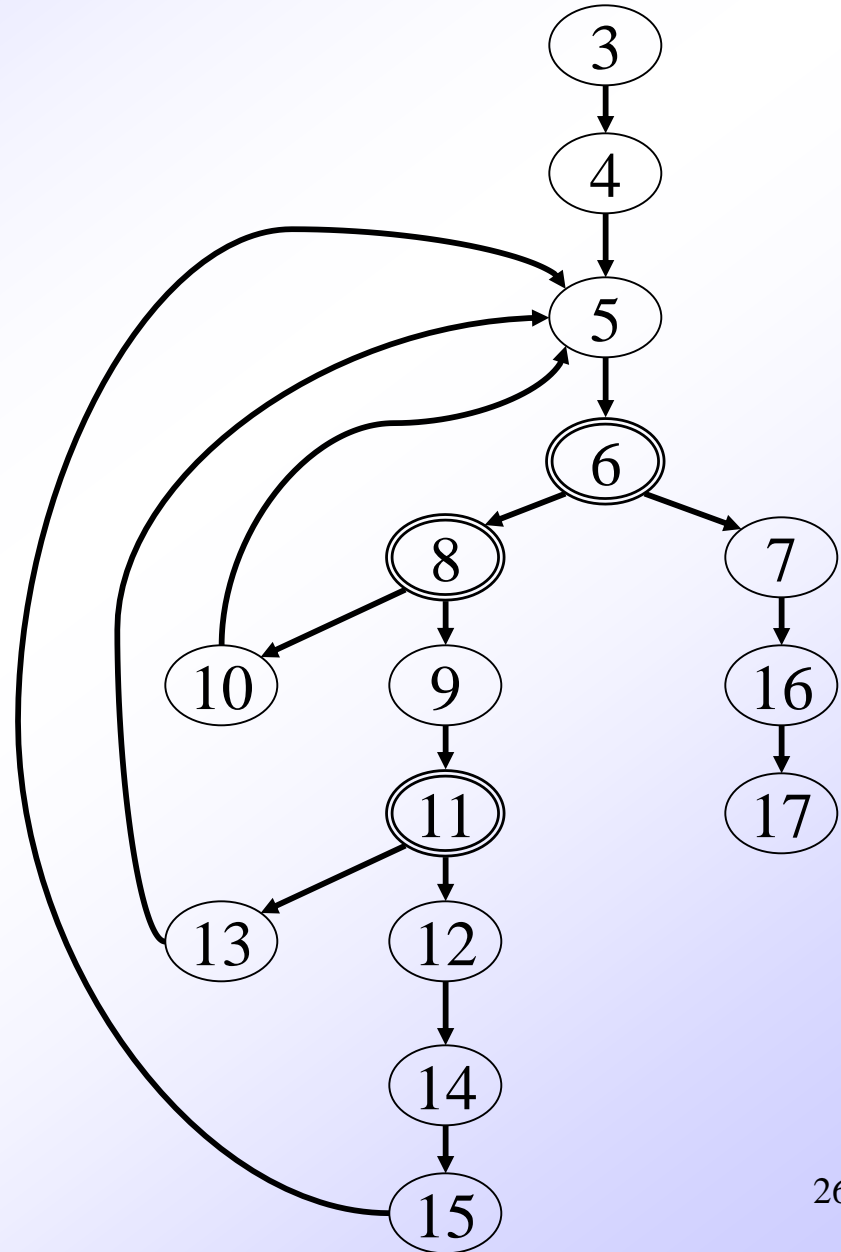
A Sample Function to Diagram and Analyze

```
 1   int functionZ(int y)
 2   {
 3   int x = 0;

 4   while (x <= (y * y))
 5       {
 6       if ((x % 11 == 0) &&
 7           (x % y == 0))
 8           {
 9           printf("%d", x);
10           x++;
11           } // End if
12       else if ((x % 7 == 0) ||
13                 (x % y == 1))
14           {
15           printf("%d", y);
16           x = x + 2;
17           } // End else
18       printf("\n");
19       } // End while

20   printf("End of list\n");
21   return 0;
22   } // End functionZ
```

A Sample Function to Diagram and Analyze

```
1    int functionZ(int y)
2    {
3    int x = 0;

4    while (x <= (y * y))
5        {
6        if ((x % 11 == 0) &&
7            (x % y == 0))
8            {
9            printf("%d", x);
10           x++;
11           } // End if
12       else if ((x % 7 == 0) ||
13                 (x % y == 1))
14           {
15           printf("%d", y);
16           x = x + 2;
17           } // End else
18       printf("\n");
19       } // End while

20   printf("End of list\n");
21   return 0;
22   } // End functionZ
```
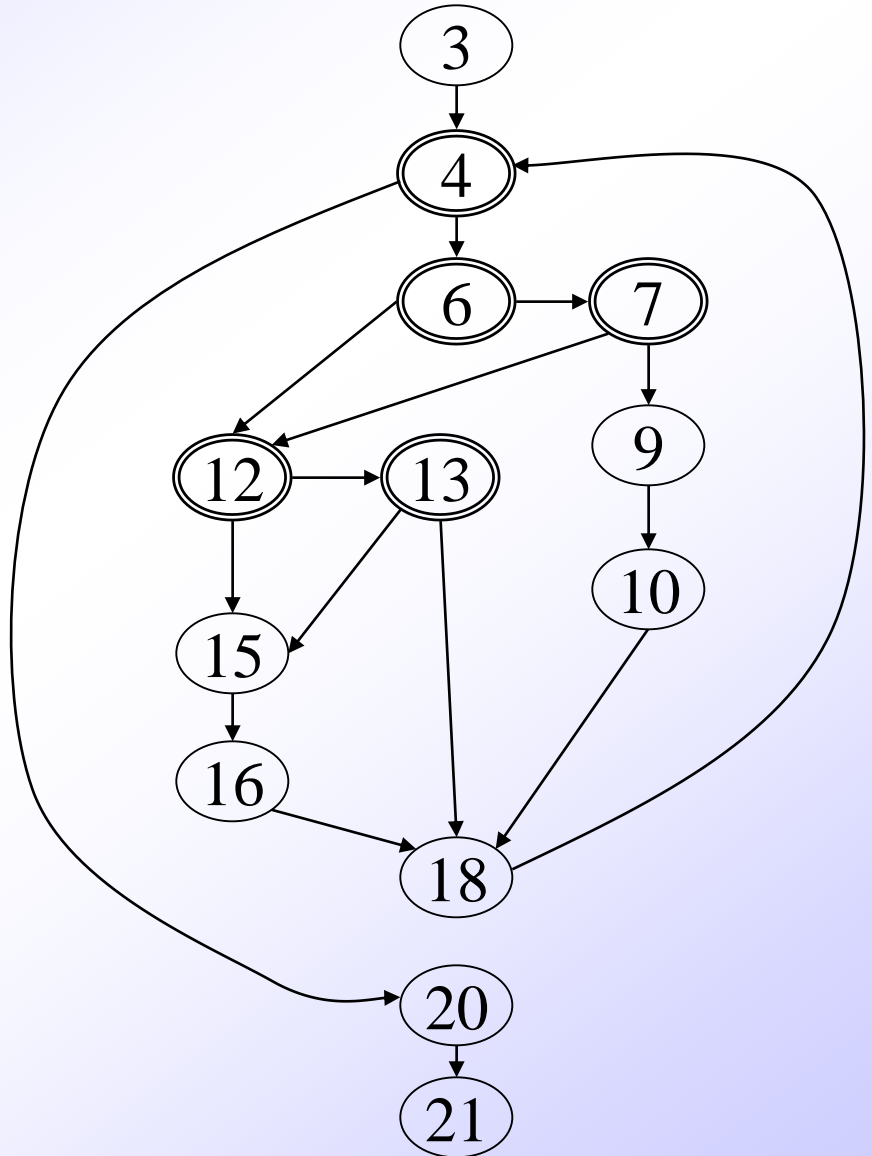
Loop Testing - General

- A white-box testing technique that focuses exclusively on the validity of loop constructs
- Four different classes of loops exist
  - Simple loops
  - Nested loops
  - Concatenated loops
  - Unstructured loops
- Testing occurs by varying the loop boundary values
  - Examples:

    ```
    for (i = 0; i < MAX_INDEX; i++)

    while (currentTemp >= MINIMUM_TEMPERATURE)
    ```

Loop Testing

Simple loop

Nested Loops

Concatenated Loops

Unstructured Loops
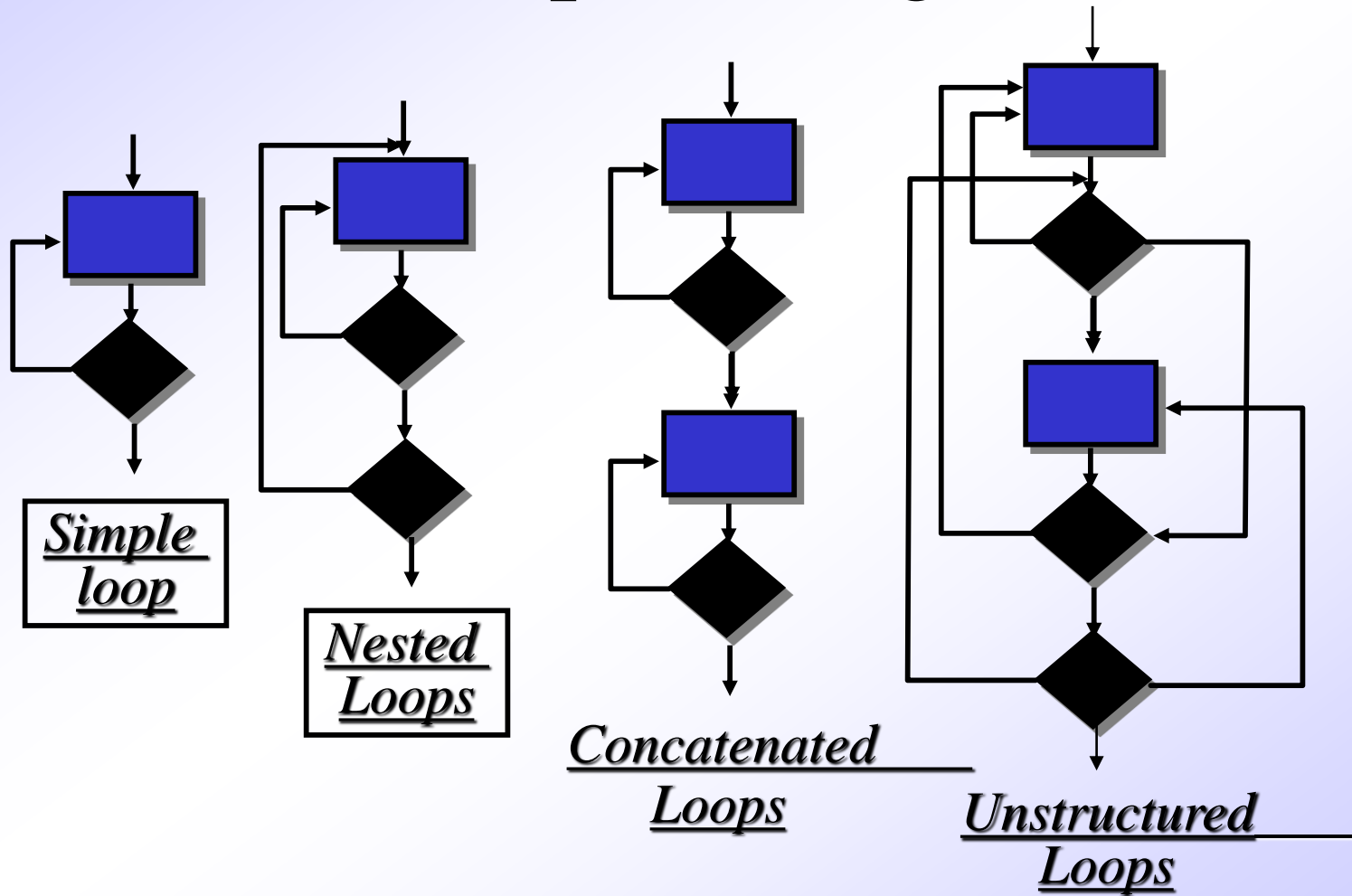
Testing of Simple Loops

1) Skip the loop entirely   i>n

2) Only one pass through the loop   i==1

3) Two passes through the loop

4) m passes through the loop, where m < n

5) n −1, n, n + 1 passes through the loop   (boundary condition value gulo test)

'n' is the maximum number of allowable passes through the loop

Testing of Nested Loops

1) Start at the innermost loop; set all other loops to minimum values

2) Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values; add other tests for out-of-range or excluded values

3) Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values

4) Continue until all loops have been tested

Testing of Concatenated Loops

- For independent loops, use the same approach as for simple loops
- Otherwise, use the approach applied for nested loops

Testing of Unstructured Loops

- Redesign the code to reflect the use of structured programming practices
- Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

Advantages Of White Box Testing

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.

- Testing is more thorough, with the possibility of covering most paths.

1.(each loop,each statement tested or executed atleast once)
2.atomic function/each module gula test hosse
3.GUI er dorkar nai cause developer ra test kortese . software bananor suru thkeei developer ra auto test korei astese

Disadvantages of white box testing

- Developers often test with the intent to prove that the code works rather than proving that it doesn't work
- Developers tend to skip the more sophisticated types of white box tests (e.g., condition testing, data flow testing, loop testing, etc.), relying mostly on code coverage
  - And developers also tend to overestimate the level of code coverage
- White box testing focuses on testing the code that's there. If something is missing, white box testing might not help you.
- There are many kinds of errors that white box testing won't find
  - Timing and concurrency bugs
  - Volume and load limitations
  - Efficiency problems
  - Usability problems

1.developers often try to prove that their code work properly rather than proving that it doesnot work
2.if any function declare kora na thake/create kora na thake tahole white box sedike kheyal korena sudhu code er statement er dike kheyal kore

Black-box Testing

Black-Box Testing

requirements

output

input

events
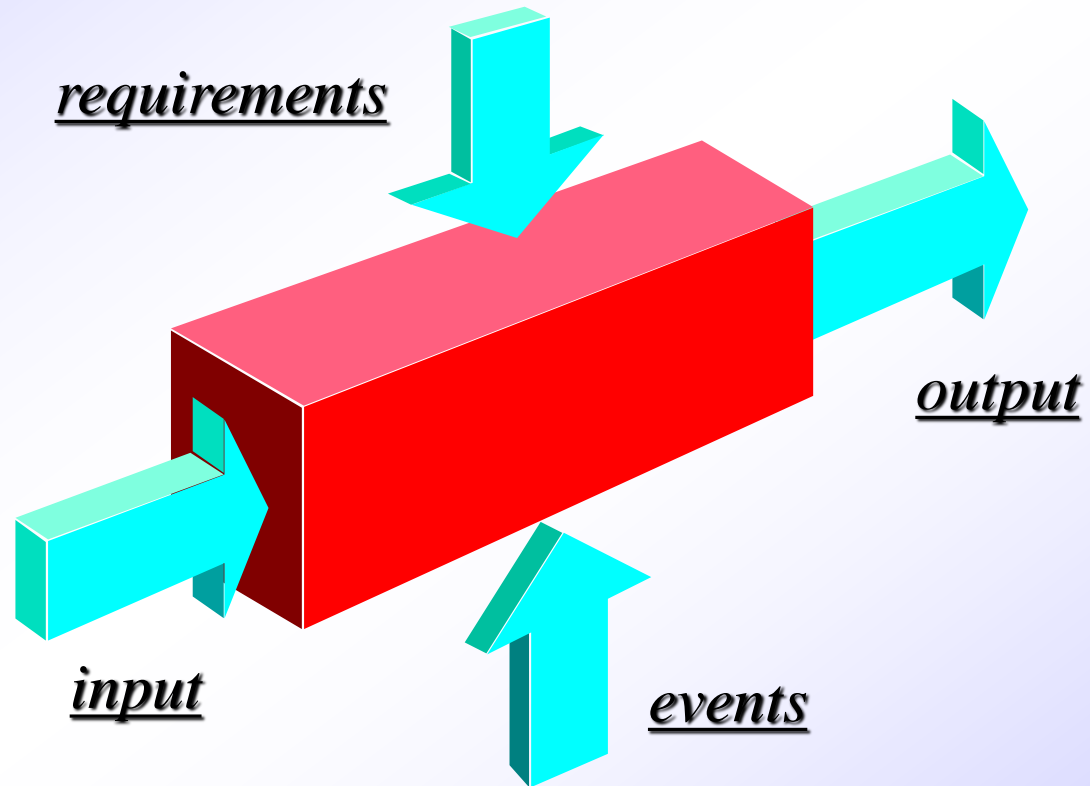
Black-box Testing

- <u>Complements</u> white-box testing by uncovering different classes of errors
- Focuses on the functional requirements and the information domain of the software
- Used during the <u>later stages</u> of testing after white box testing has been performed
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- The test cases satisfy the following:
  - Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - Tell us something about the presence or absence of <u>classes of errors</u>, rather than an error associated only with the specific task at hand

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

Questions answered by Black-box Testing

- How is functional validity tested?
- How are 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 boundary values 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 Methods

- Software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.
- To accomplish these steps, the software engineer begins by creating
  - a *graph*—a collection of *nodes* that represent objects;
  - *links* that represent the relationships between objects;
  - *node weights* that describe the properties of a node (e.g., a specific data value or state behavior); and
  - *link weights* that describe some characteristic of a link.
  - A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction.
  - A *bidirectional link,* also called a *symmetric link*, implies that the relationship applies in both directions.
  - *Parallel links* are used when a number of different relationships are established between graph nodes.

Graph-Based Testing Methods

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

Object #1 — Directed link (link weight) → Object #2

Undirected link

Node weight (value)

Object #3

Parallel links (editing)

(A)

43
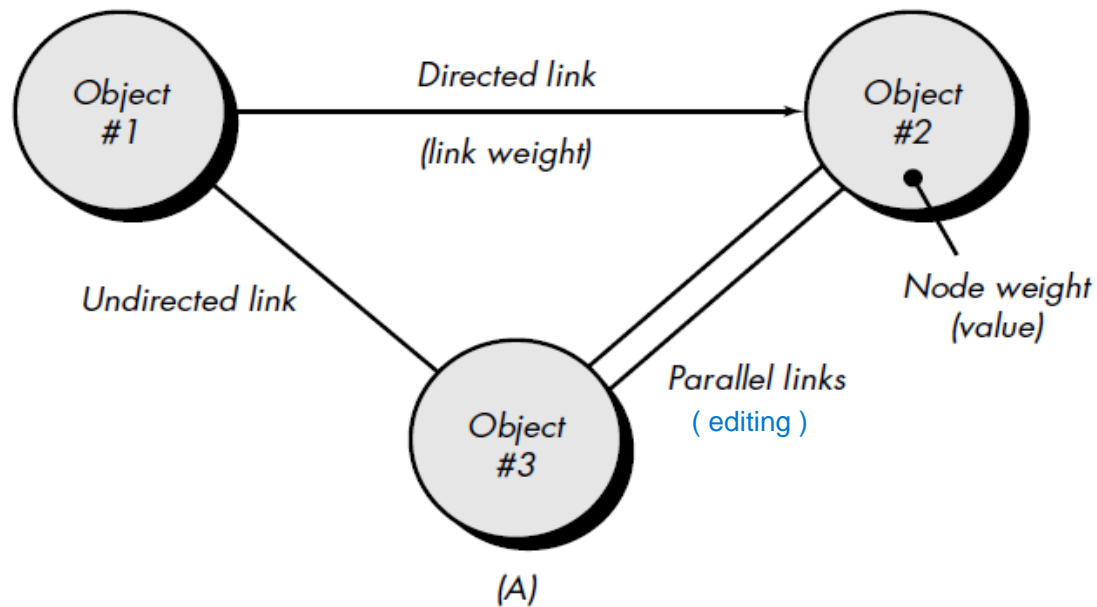
Graph-Based Testing Methods

(Notepad example) :

New file menu select

Menu select generates

(generation time < 1.0 sec)

Document window

Is represented as

Allows editing of

Document text

Contains

Attributes:
Start dimension: default setting
or preferences
Background color: white
Text color: default color
or preferences

(B)
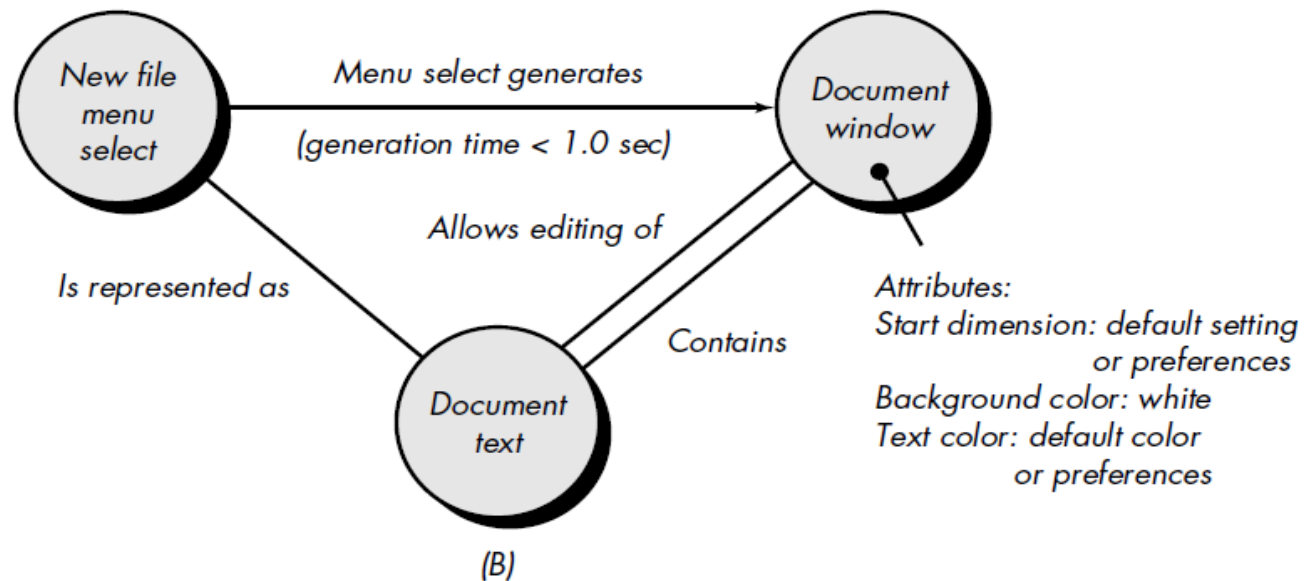
Graph-Based Testing Methods

- *Object #1* = **new file menu select**
- *Object #2* = **document window**
- *Object #3* = **document text**

- A menu select on **new file** generates a **document window.**

- The node weight of **document window** indicates that the Window must be generated in less than 1.0 second.

- An undirected link establishes a symmetric relationship between the **new file menu select** and **document text,** and parallel links indicate relationships between **document window** and **document text.**

- The software engineer then derives test cases by traversing the graph and covering each of the relationships shown.

Equivalence Partitioning

Classes

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived

- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed

- Test case design is based on an evaluation of equivalence classes for an input condition

- An equivalence class represents a set of valid or invalid states for input conditions

- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once

Guidelines for Defining Equivalence Classes

- If an input condition specifies <mark>a range</mark>, one valid and two invalid equivalence classes are defined

  valid   invalid   invalid

  – <mark>Input range</mark>: 1 – 10   <mark>Eq classes:</mark> $\{1..10\}, \{x < 1\}, \{x > 10\}$

- If an input condition requires <mark>a specific value</mark>, one valid and two invalid equivalence classes are defined

  – <mark>Input value</mark>: 250   <mark>Eq classes:</mark> $\{250\}, \{x < 250\}, \{x > 250\}$

- If an input condition specifies <mark>a member of a set</mark>, one valid and one invalid equivalence class are defined

  – <mark>Input set:</mark> $\{-2.5, 7.3, 8.4\}$ <mark>Eq classes:</mark> $\{-2.5, 7.3, 8.4\}, \{$any other $x\}$

- If an input condition is <mark>a Boolean value</mark>, one valid and one invalid class are define

  – <mark>Input:</mark> {true condition}   <mark>Eq classes:</mark> {true condition}, {false condition}

Boundary Value Analysis

- A greater number of errors occur at the <u>boundaries</u> of the input domain rather than in the "center"

- Boundary value analysis is a test case design method that <u>complements</u> equivalence partitioning
  - It selects test cases at the <u>edges</u> of a class
  - It derives test cases from both the input domain and output domain

Guidelines for Boundary Value Analysis

- 1. If an input condition specifies a <u>range</u> bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*

- 2. If an input condition specifies a <u>number of values</u>, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested

- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above

- If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

Example

- Example: For a unit processed normally when input data is between 10 and 30

. . .	7	8	9	10	11	. . .	29	30	31	32	33	. . .
Invalid equivalence class				Valid equivalence class					Invalid equivalence class			

- Equivalence partitioning: {7, 21, 39} ... One representative value from each class, Boundary value analysis: {9, 10, 30, 31} ... All boundary values from each class

Advantages / Pros of Black Box Testing

- Unbiased tests because the designer and tester work independently

- Tester is free from any pressure of knowledge of specific programming languages to test the reliability and functionality of an application / software

- Facilitates identification of contradictions and vagueness in functional specifications

- Test is performed from a user's point-of-view and not of the designer's

- Test cases can be designed immediately after the completion of specifications

Disadvantages / Cons of Black Box Testing

- Tests can be redundant if already run by the software designer

- Test cases are extremely difficult to be designed without clear and concise specifications

- Testing every possible input stream is not possible because it is time-consuming and this would eventually leave many program paths untested

- Results might be overestimated at times

- Cannot be used for testing complex segments of code

customer er requirement frequently change hole bar bar blackbox testing kora costly

White box vs black box testing

Black Box Testing	White Box Testing
1. Black box testing techniques are also called functional testing techniques.	1. White box testing techniques are also called structural testing techniques.
2 Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester	2. White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
3. It is mainly applicable to higher levels of testing such as Acceptance Testing and System Testing	3. Mainly applicable to lower levels of testing such as Unit Testing and Integration Testing
4. Black box testing is generally done by Software Testers	4. White box testing is generally done by Software Developers
5. Programming knowledge is not required	5. Programming knowledge is required
6. Implementation knowledge is not required.	6. Implementation knowledge is required

Thank you