Software Testing

Why Testing and Analysis?

- **Software is never correct no matter which developing technique is used**
- **Any software must be verified.**
- **Software testing and analysis are:**
  - important to control the quality of the product (and of the process)
  - very (often too) expensive
  - difficult and stimulating

Real life examples

- First U.S. space mission to Venus failed. (reason: missing comma in a Fortran do loop)
- December 1995: AA, Boeing 575, mountain crash in Colombia, 159 killed. (Incorrect one-letter computer command)
- June 1996: Ariane-5 space rocket, self-destruction, $500 million. (reason: reuse of software from Ariane-4 without recommended testing).

Testing Strategies

SW testing is one element of a broader topic referred to as V&V (many SQA activities).

- **Verification** (correctly implement specific function)
  - Defect detection and correction
  - Comparison between implementation and the corresponding specification
  - Are we building the product right?
- **Validation** (SW is traceable to customer requirements)
  - Defect prevention
  - Provision of sound basis for specific design decisions
  - Are we building the right product?

Example: Checking the printing of receipts is verification and correct printing (info) is validation.

Types/Strategies of testing

- **Code Inspections**
- **Software (Module) Testing**
  - Unit Testing
  - Functional Testing
- **Integration Testing**
  - Compliance
  - Interoperability Testing
- **System Testing**
  - Recovery
  - Security
  - Load / Stress
- **Acceptance Testing**
- **Stress, Security, performance, Load Testing etc**

Software Testing

- **Objectives (different for each type)**
  (1) it does what it should ---- **requirements conformance**
  (2) it does it well and efficiently ------ **performance**
  (3) it does so without error ------ **errors-correctness**
  (4) And so on ------ **an indication of quality**
- Carefully designed and coded program - less chances of errors but some always exist.
- To some people: "A process of checking a program to show there are no errors"- implies that a successful testing is one that shows no error.
- This a **wrong approach**. The aim of testing is to find errors so that they can be corrected.

Software Testing

- A better definition: ”the process of running software with the intent of finding an error”.

- The way most testing works is to input a set of values, then compare the expected result with the out comes/computed ones.

- Depending on the type of data input, we can identify three levels of program correctness possible, probable, and absolute correctness.

Three Levels of Correctness

Possible correctness

- Obtaining correct output for some arbitrary input (single set).
- If the outcome is wrong, the program cannot possibly be correct.
- For example a multiply program: for input 2 and 3, if result is 6; is possibly correct.

Probable correctness

- Obtaining correct output for a number of carefully selected inputs.
- If all potential problematic areas are checked in this way, the program is probably correct.
- Try several values, including the obvious “problem” values such as zero, largest negative, 1, -1 and so on.

Absolute correctness

- can be demonstrated only by a test that involves every possible combination of inputs.
- Requires huge amount of time; it is therefore not practical {However for some programs we can prove correctness mathematically}.
- Imagine the same above test for a 32 bit machine (how many combinations are possible? (Hundreds of Millions).

Testability

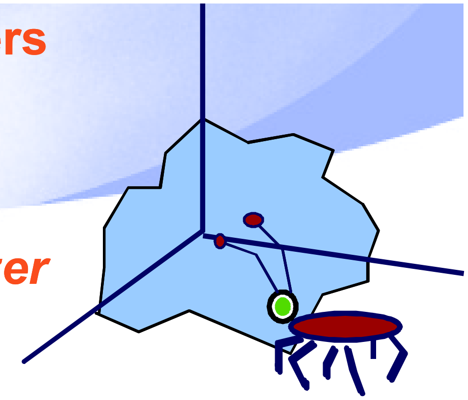
Which S/W is testable? Characteristics of a testable S/W.

- 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 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

- A successful test -- -- that uncovers an as-yet undiscovered error.
- Minimum number of required tests with 100% functional coverage and 0% redundancy.
- Rich variety of test case design methods
  - Cause-effect graphing, Equivalence Class Partitioning, Boundary Analysis, and vendor specific: client/server, OO test case design.

Possible approaches:

(1) Knowing the specified functions, tests can be conducted to demonstrate that each function is fully operational (**Black Box**).

(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh" (**White Box**).

Why bother with white box testing?

Black box testing:

- **Requirements fulfilled**
- **Interfaces available and working**

But what about

- **the internal structure of a component,**
- **interactions between objects?**

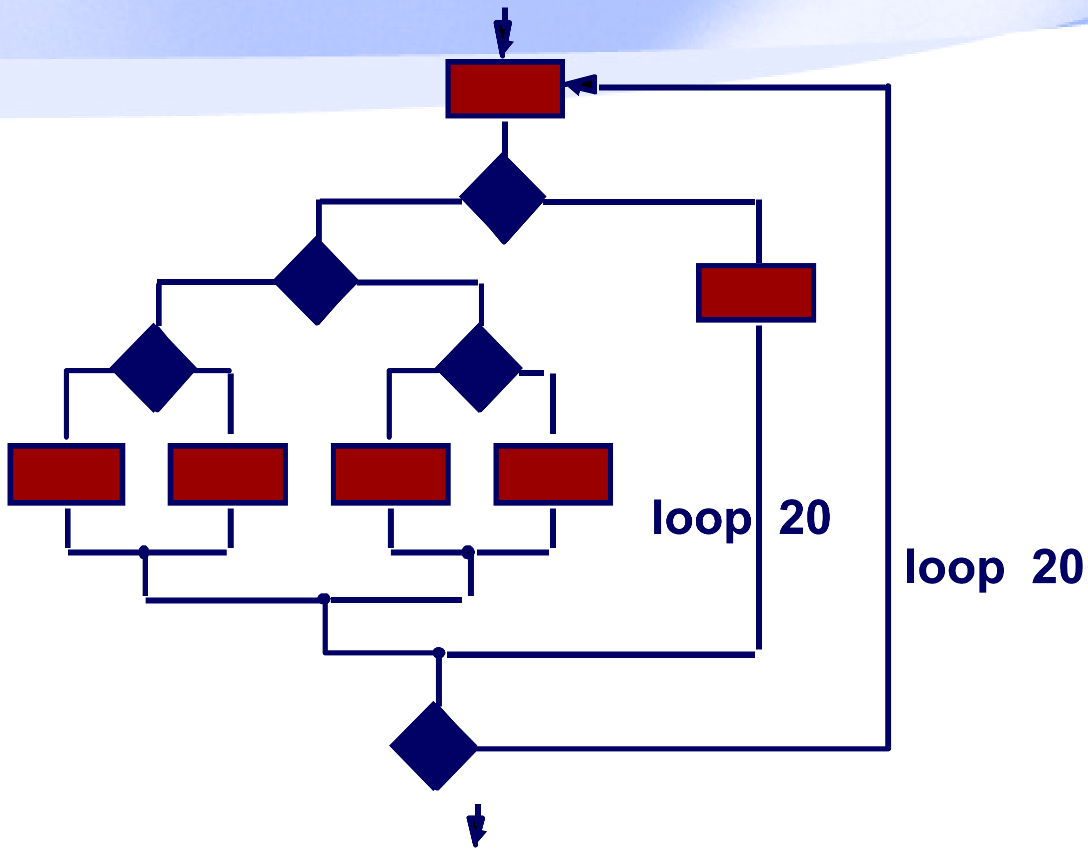
☞ **white box testing**

FURTHER

- Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- Sometime a path executes counterintuitive.
- Typographical errors are random.

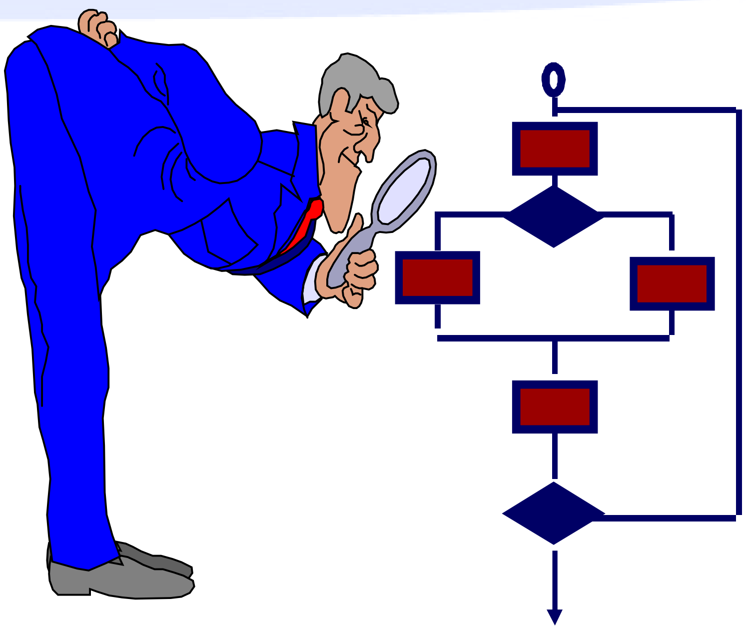
Also: if black box testing finds error, locating it is easier with additional white box testing!

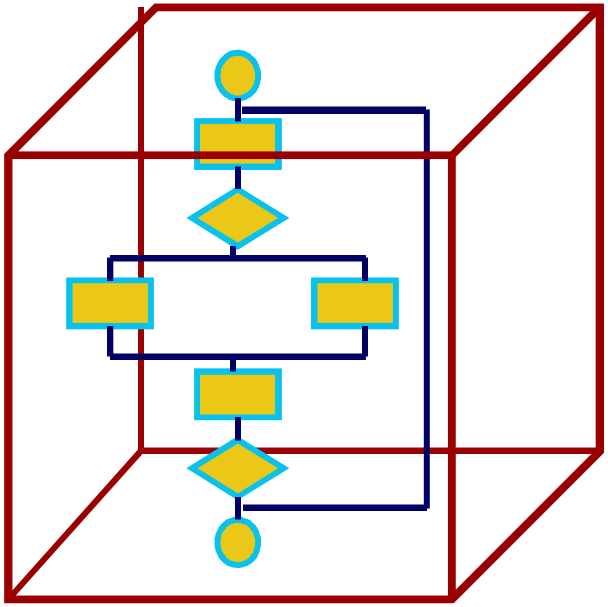
Exhaustive Testing



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

White-Box Testing





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

White-Box

- White box testing, sometimes called glass-box and structural testing.
- Various aspects like Statement Coverage Criteria, Edge coverage, Condition Coverage, Path Coverage are defined mathematically and test set is designed accordingly.
- There are no algorithms for generating white box test data. However the check list might help:
  - Has every line of code been executed at least once by test data.
  - Have all default paths been traversed at least once.
  - Have all significant combinations of multiple conditions been identified and
  - Have all logical decisions exercised on their true and false sides.
  - Have all loops executed at their boundaries and within their operational bounds.
  - Have internal data structures validated▪

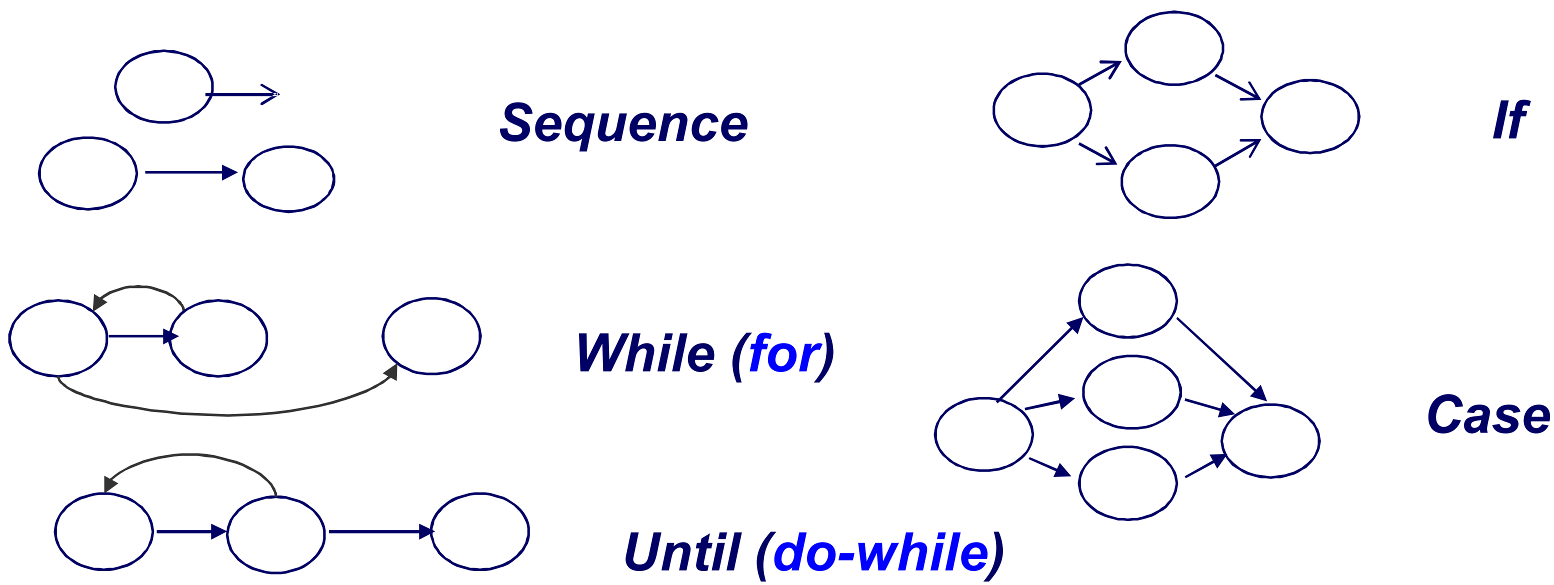
Basis Path Testing: A White Box technique

- This method enables the designer to derive a logical complexity measure of a procedural design and use it as a guide for defining a ***basis set*** of **execution paths**.
- Guarantees to execute every statement in the program at least once.

Basis Path Testing

FLOW GRAPH Notation (nodes, links)

Each **circle (node)** represents **One or more** non-branching PDL or source code statements)

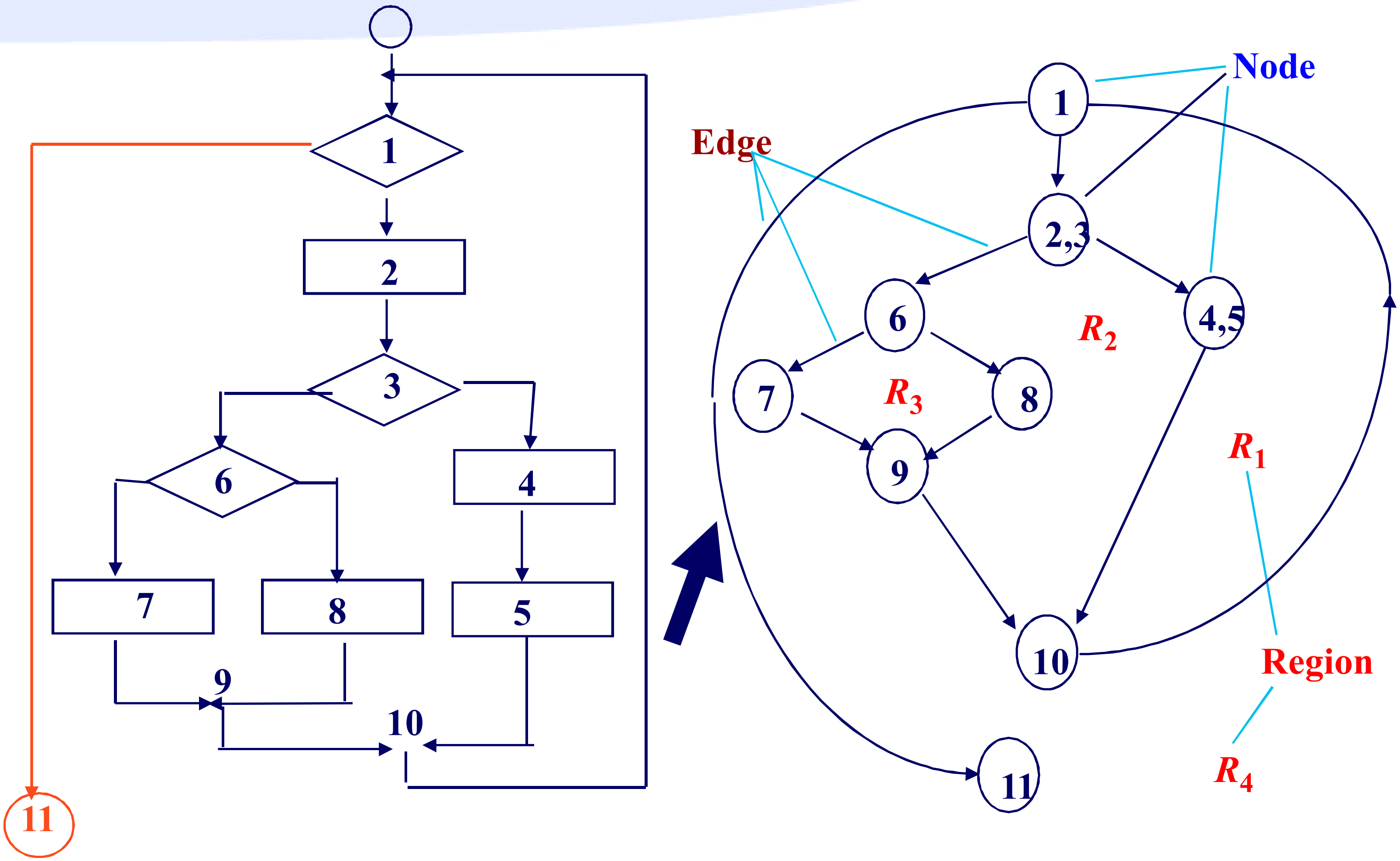


- An arrow (link) represents control flow.

Nodes, Edges, And Regions

- Each circle is a node (represents one or more procedural statements -- sequence of process boxes and a decision diamond can map into a single node)
- A conditional node is called a predicate node
- Arrows are called edges and must terminate at a node.
- Areas bounded by edges and nodes are called regions (including the area outside the graph)

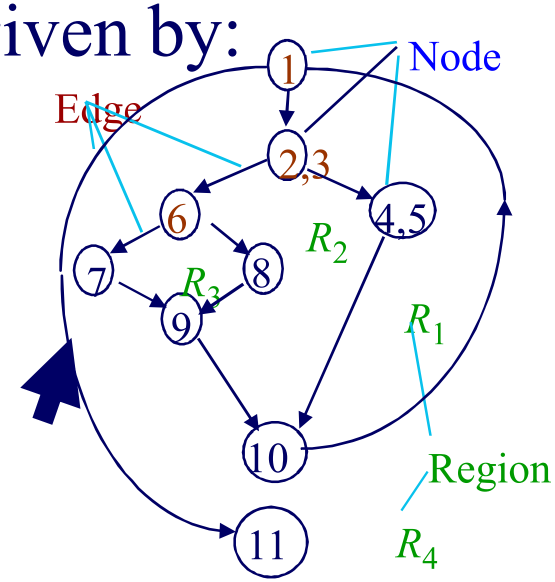
Flowchart And Flow Graph



Cyclomatic Complexity

- Cyclomatic complexity provides a quantitative measure of the logical complexity of a program
  - It is the number of tests that must be conducted to assure that all statements have been executed at least once
- Cyclomatic complexity, V(G) is given by:

(1) V(G) = E - N + 2
where ***E*** *is the number of* ***edges*** *and* ***N*** *number of* ***nodes*** *(11-9 +2 =* **4***)*
(2) V(G) = P + 1, where ***P*** *is the number of* ***predicate nodes*** (3+1 = **4**)
(3) V(G) = number of regions (**4**)

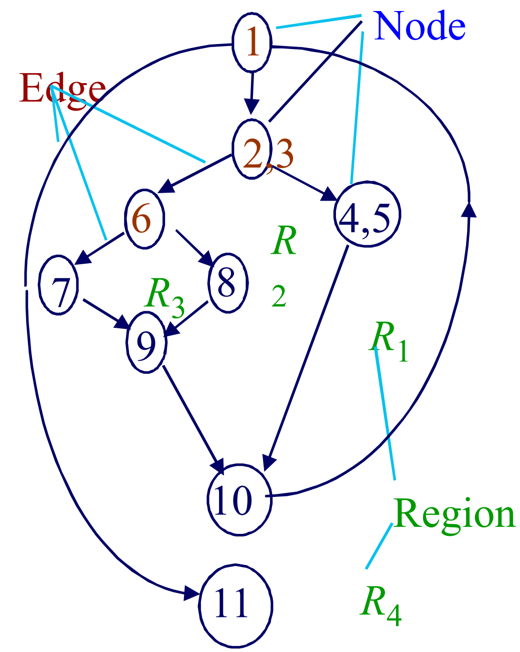


Independent Paths

- The value for cyclomatic complexity defines the number of independent paths in the basis set of a program
- An independent path is 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
- A **set of independent paths** for a flow graph composes a **basis set.**

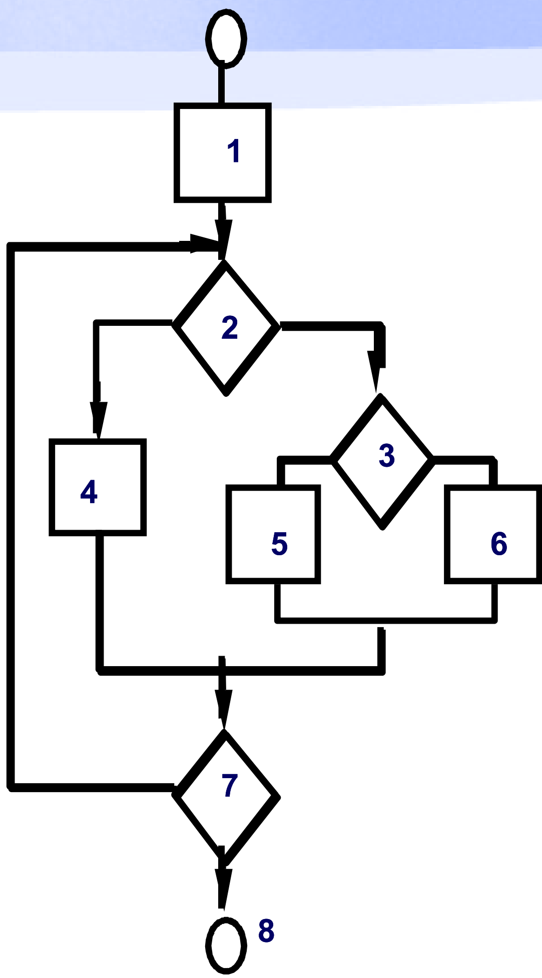
Deriving Test Cases

- Using the design or code, draw a corresponding flow graph
  you don't need a flow chart,
- Determine the cyclomatic complexity (even without developing flow graph: count each simple logical test (compound tests as 2 or more) + 1
- Determine a basis set of paths
- Prepare test cases that will force execution of paths in the basis set
- basis path testing should be applied to critical modules



Exercise: Read and understand the example (Figure 17.4 Pressman5th –450, 6th – Fig

Basis Path Testing



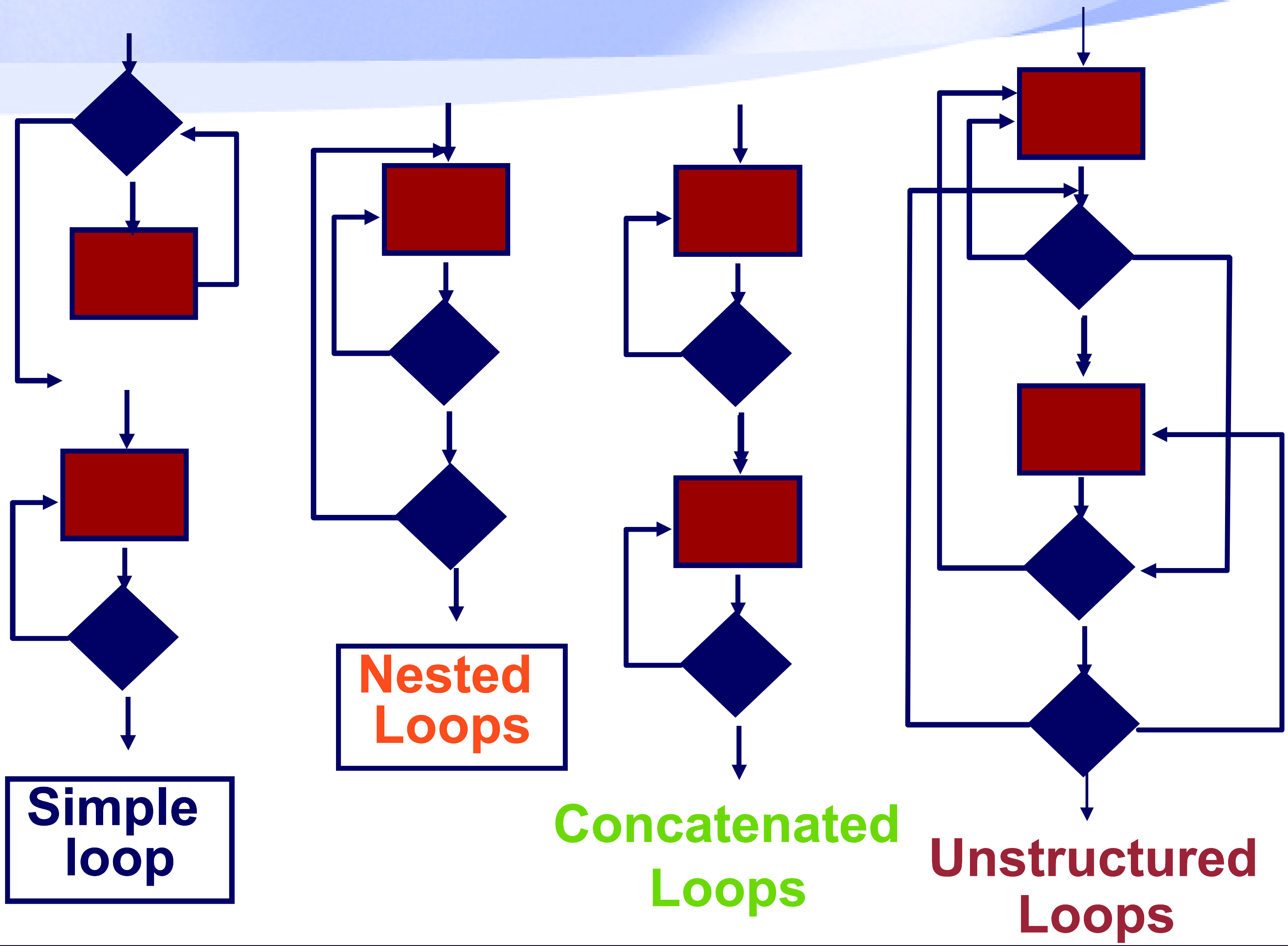
derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

derive test cases to exercise these paths.

Control Structure Testing: Loop



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

Loop Testing: Nested Loops

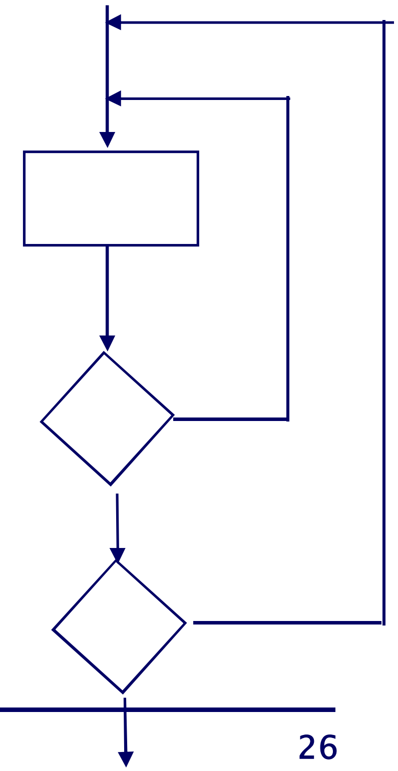
Nested Loops

(1) Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

(2) Test the min+1, typical, max-1 and max for the innermost loop (while holding the outer loops at their minimum values).

(3) Move out one loop and set it up as in step 2, holding all other loops at typical values.

Continue this step until the outermost loop has been tested.



Concatenated Loops

If the loops are independent of one another
  then treat each as a simple loop
else
  treat as nested* loops
endif

Theoretical Foundation of Testing

- **Mathematical model**: Let P be a program, and let D and R denote its input (Data) and output (Result) ranges. That is, D is a set of all data that can correctly be supplied to P, and the results of P's execution, if any, are elements of R.

TEST EXAMPLE : **Statement Coverage Criteria**: We can use above model to define test as: Select a test set T such that, by executing P for each d (subset of D) in T, each elementary statement of P is executed at least once.

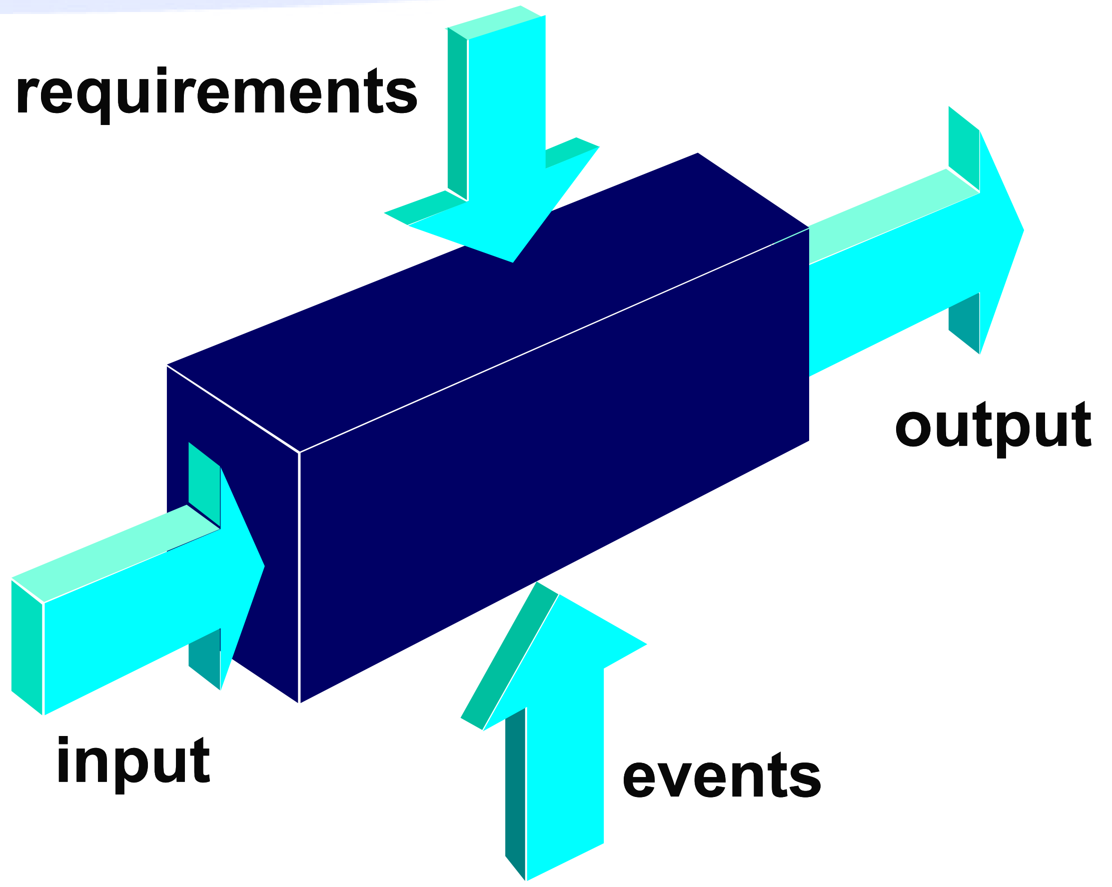
Test Case Design Examples

- Number of test cases in a test does not necessarily contribute: To test this incorrect program

```
If x > y Then
        max = x;
else
        max = x;
endif;
```

- The test case set [x =3, y= 2; x= 2, y= 3] is able to detect the error. Whereas
- Test case set [x=3, y = 2; x =4, y = 3, x = 5, y = 1] is not, although it contains more test cases.

Black-Box Testing



Black Box Testing

- Types of errors regarding functional requirements of software:

  -- Incorrect or missing functions

  -- Interface errors

  -- Error in data structure & external data base access

  -- Performance errors

  -- Initialization & termination errors
- No functional requirements NO Black Box Testing.
- Demonstrates that each function is fully operational.
- Uncovers different kind of errors than white box testing.
- Performed later in the testing process.

- Black box techniques derive a set of test cases that satisfy the following criteria:

  (1) Test cases reduce the number of additional test cases that must be designed to achieve reasonable testing

  (2) Test cases that tell us something about the presence or absence of classes of errors, rather than errors associated only with the specific test at hand.

- Black box techniques can supplement the test cases generated by white box.

How to Design Test Cases?

- ✓ Equivalence class partitioning
- ✓ Boundary value analysis
- ➢ Cause / effect graphing (for combinations of input conditions)
- ✓ Error Guessing

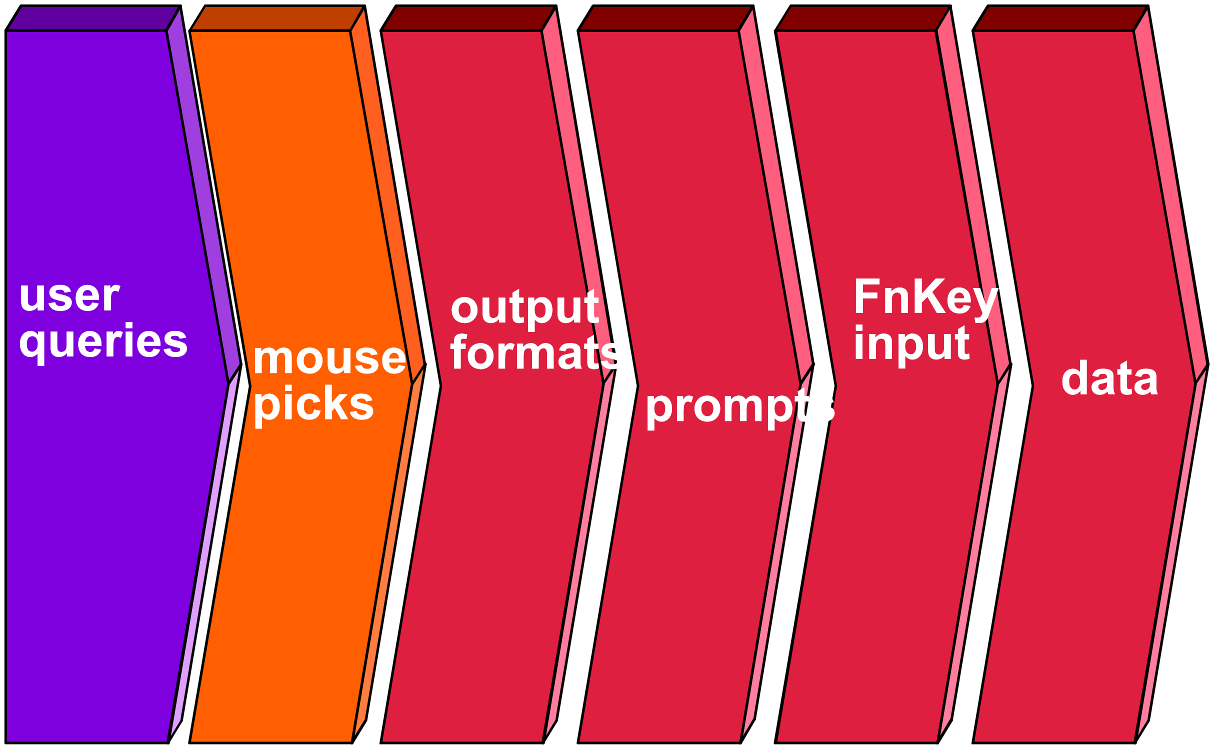
Equivalence Class Partitioning

Two considerations:

1. each test case should invoke as many different input conditions as possible in order to minimize the total number of test cases necessary.
2. one should try to partition the input domain of a program into a finite number of equivalence classes.

- **Equivalence partitioning uses the idea of equivalence classes.**
- **An equivalence class is a set of data which as for specification is concerned will be treated identically (equivalently). ***
- **Objective is to identify those classes of data, which will cause the module to respond in a different manner from other classes.**
- **This is done by reading the specification and creating a list of all characteristics of programs (e.g. must be numeric, two integers are input).**

Equivalence Partitioning



Identifying Equivalence classes

- A key concept in the identification of classes is negation, i.e. If a characteristic is identified as an equivalence class, then one should immediately negate the characteristic in order to find examples of classes which should cause the module to do something different such as “generate an error message”.
- Partitioning each input condition into two or more groups.
- Two types of equivalence classes are identified:
  1. Valid Equivalence Classes

  2. Invalid Equivalence Classes

Sample Equivalence Classes

Valid data

- user supplied commands
- responses to system prompts
- file names
- computational data
  - physical parameters
  - bounding values
  - initiation values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

Invalid data

- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place

Defining equivalence classes

- Input condition is a range: one valid and two invalid classes are defined
- Input condition requires specific value: one valid and two invalid classes are defined
- Input condition is boolean: one valid and one invalid class are defined

☞ Then define one test case for each equivalence class

Equivalence classes Examples

Input or Output Event	Valid Equivalence Classes	Invalid Equivalence Classes
Enter a non-zero digit	1 – 9	< 1, > 9 Letters and other non-numeric characters
Enter the first letter of a name	First character is a capital letter First character is a lower case letter	First character is not a letter
Draw a line **Max. 4 inches**	From 1 to 4 inches long	No line Longer than 4 inches Not a line (a curve)

Characteristic	**Valid equivalence class**	**Invalid equivalence class**
First char must be alphabetic	Letter	Non-letter
Next three numerics	All numeric	One char non-numeric
Range 100-500	In range	(i) Above range (ii) Below range

Equivalence class table for customer-acc-number

Exercise: Create similar table for your roll number

Boundary-value Analysis

Boundary-value analysis differs from Equivalence Partitioning in two respects:

1. Rather than selecting any element to represent an equivalence class, boundary-value analysis requires that one or more elements be selected such that each edge of the equivalence class is subjected to a test.
2. Rather than focusing attention on the input conditions (input space), test cases are also derived by considering the result space (i.e., output equivalence classes)

- It will be more exacting to create test cases for number = 99, 100, 500, 501, rather than numbers = 50, 250, 900 for previous example.
- if a module uses a file of records, how will program react if there are no records on the file; also if a transaction file is used for updating a master file; have all permutations of EOF conditions been considered.
- If a module is passed an array, what if it contains zero elements? when it contains maximum number of elements, and so on a pointer to an array accessing out side an array-boundary.

Error-Guessing

- Some people design the test cases by intuition and experience.

- The basic idea is to enumerate a list of possible errors and then write test cases based on the list.

Testing Phase

What content should be included in a software test plan?

- Testing activities and schedule
- Testing tasks and assignments
- Selected test strategy and test models
- Test methods and criteria
- Required test tools and environment
- Problem tracking and reporting
- Test cost estimation

Test Execution

- using manual approach
- using an automatic approach
- using a semi-automatic approach

Basic activities in test execution:

- Select a test case
- Set up the pre-conditions for a test case
- Set up test data
- Run a test case following its procedure
- Track the test operations and results
- Monitor the post-conditions of a test case & expected results
- Verify the test results and report the problems if there is any
- Record each test execution

Example: *Windows Calculator*

R-001:
The users should be able to add two numbers and view their result on the display.

Use Case: *UC01*	Add Two Numbers
Actors:	**User**
Purpose:	**Add two numbers and view their result**
Overview:	**The user inputs two numbers and (then adds them and) checks the result, displayed on the screen.**
Type:	**Primary, Real**
Cross References:	**R-001**

Typical Course of Events

Actor Action	System Response
1. The actor opens the calculator. The keypad and display screen appears.	
2. The actor input the first number by clicking on the keypad or using keyboard.	3. The digit is displayed on the screen.
4. The actor clicks or presses the "+" key.	
5.The actor then adds the second number as (2).	6. The pressed digit is displayed on the screen.
7. The actor clicks the "=" key.	8. The sum of the two digits is displayed on the screen.

Test Case

Test Case ID: *T-101*　　***Test Item:*** *Add Numbers*
Wrote By: *(tester name)*　*Junaid*　***Documented Date:*** *26th April 2005*
Test Type: *Manual*　　***Test Suite#:*** *NA*
Product Name: *Windows Calculator*　　***Release and Version No.:*** *V 1.0*
Test case description:
Add any two Numbers
Operation procedure:
Open Calculator
Press "1"
Press "+"
Press "2"
Press "="
Pre-conditions:　　***Post-conditions:***
Calculator Opened　　*Result Displayed*
Inputs data and/or events:　　***Expected output data and/or events:***
1 + 2 =　　*3*
Required test scripts (for auto): *NA*
Cross References: (Requirements or Use Cases)　***R-001, UC-01***

Bug Report

Problem ID: *B-101* ***current software name:*** *Windows Calculator*
Release and Version No.: *V 1.0*
Test type: *Manual* ***Reported by:*** *Junaid*
Reported date: *26th April 2005* ***Test case ID:*** *T-101*
Subsystem (or module name): *Calculation* ***Feature Name (or Subject):*** *Add Numbers*
Problem type (REQ, Design, Coding,): *Coding*
Problem severity (Fatal, Major, Minor,): *Major*
Problem summary and detailed description:
On adding two numbers the result is not correct.
Cause analysis: *NA*
How to reproduce? ***Why Required:***
Attachments: Nil