Software Testing

Module-1 : Basics of Software Testing

By,

Dr. Manjunath T. N.
Professor
Dept. of Information Science & Engg.
BMS Institute of Technology, Bengaluru.

Definition

- Testing is the process of executing a program with the intent of finding *errors*
- *Reasons for testing*
  - *To discover problems*
  - *To make judgment about quality or acceptability*

Definition

- Testing is obviously concerned with
  - Errors
  - Faults
  - Failures
  - Incidents

Definition

- Errors
  - ✓Synonym mistake
  - ✓Mistakes while coding-bugs
  - ✓Tend to propagate
- Fault
  - ✓ Synonym defect
  - ✓ Result/representation of error
  - ✓ Modes of expression
    - Dataflow diagram
    - Hierarchy charts
    - Narrative text
    - Source code

Definition

- ✓ Fault of commission-occurs when we enter something into a representation that is incorrect
- ✓ Fault of omission-occurs when we fail to enter correct information.

- Failure
  - ✓ Occurs when fault executes
  - ✓ Applicable to only faults of omission
- Incident
  - ✓ Symptom associated with a failure
  - ✓ Alerts user to occurrence of a failure

Definition

- Test

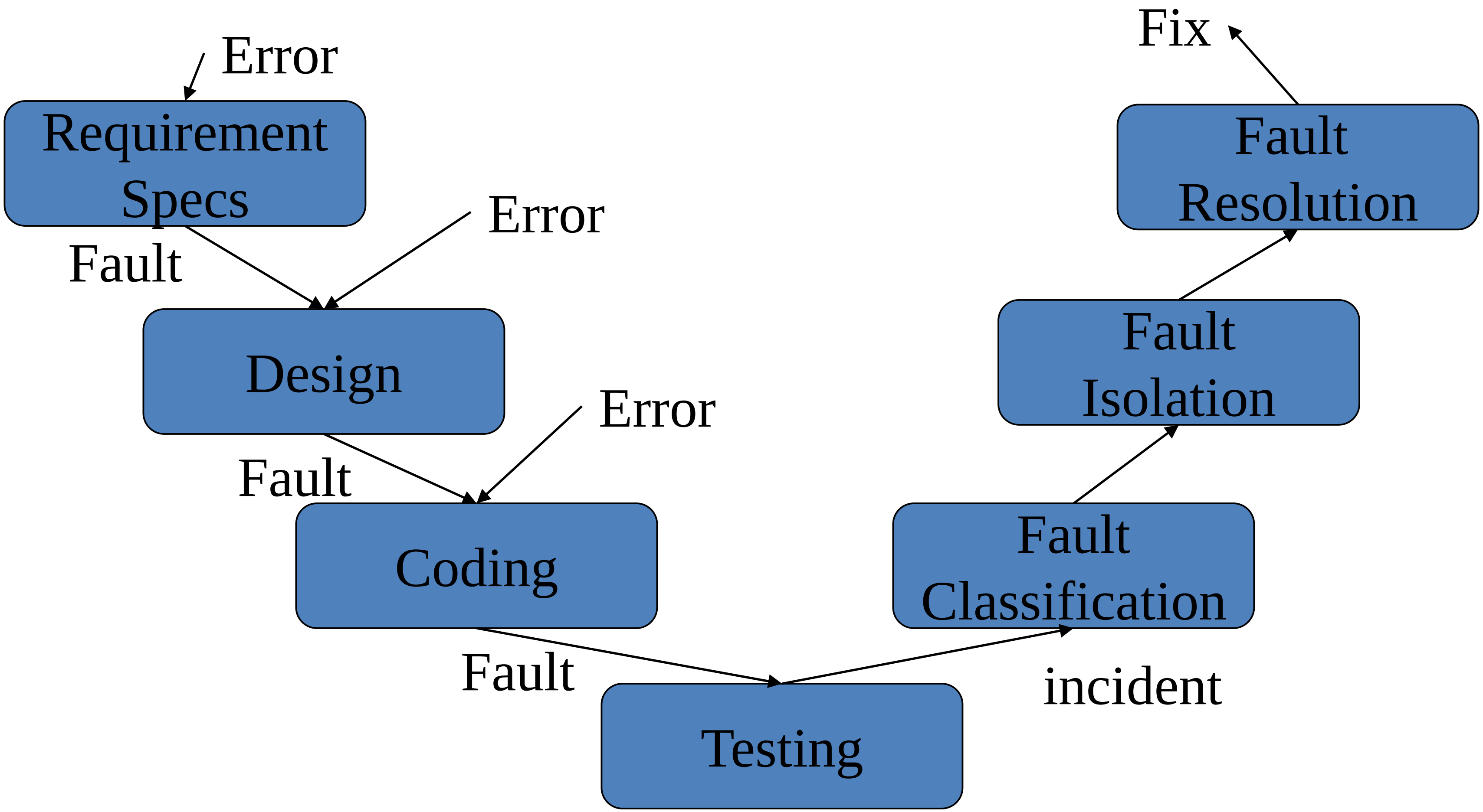
  the act of exercising software with *test cases* with an objective of

  - ✓Finding failure
  - ✓Demonstrate correct execution

- Test case

  - ✓ Has set of inputs and expected outputs.
  - ✓ Has Identity associated with program behavior

A Testing Life Cycle



A Testing Life Cycle

- Errors-faults-failures propagates in development phases.
- Tester summarises life cycle as 3 phases
  - ✓ Putting bugs IN
  - ✓ Testing phase –finding bugs
  - ✓ Getting bugs OUT
- Testing occupies central position & subdivided into
  - ✓ Test planning
  - ✓ Test case development
  - ✓ Running test cases
  - ✓ Evaluating test results.

Test cases

- Determine test cases for the item to be tested.
- Have identity- reason for being
- Inputs
  - Preconditions
  - Actual inputs
- Expected outputs
  - Actual outputs
  - Post conditions

Test cases

- Act of testing entails
  - ✓ Establishing necessary preconditions
  - ✓ Providing the test case inputs
  - ✓ Observing the outputs
  - ✓ Comparing with the expected outputs
  - ✓ Ensuring the existence of expected preconditions
- Records the execution history of test cases
  - ✓ When & by whom it was run
  - ✓ Pass/fail results
  - ✓ Version of software

Test cases

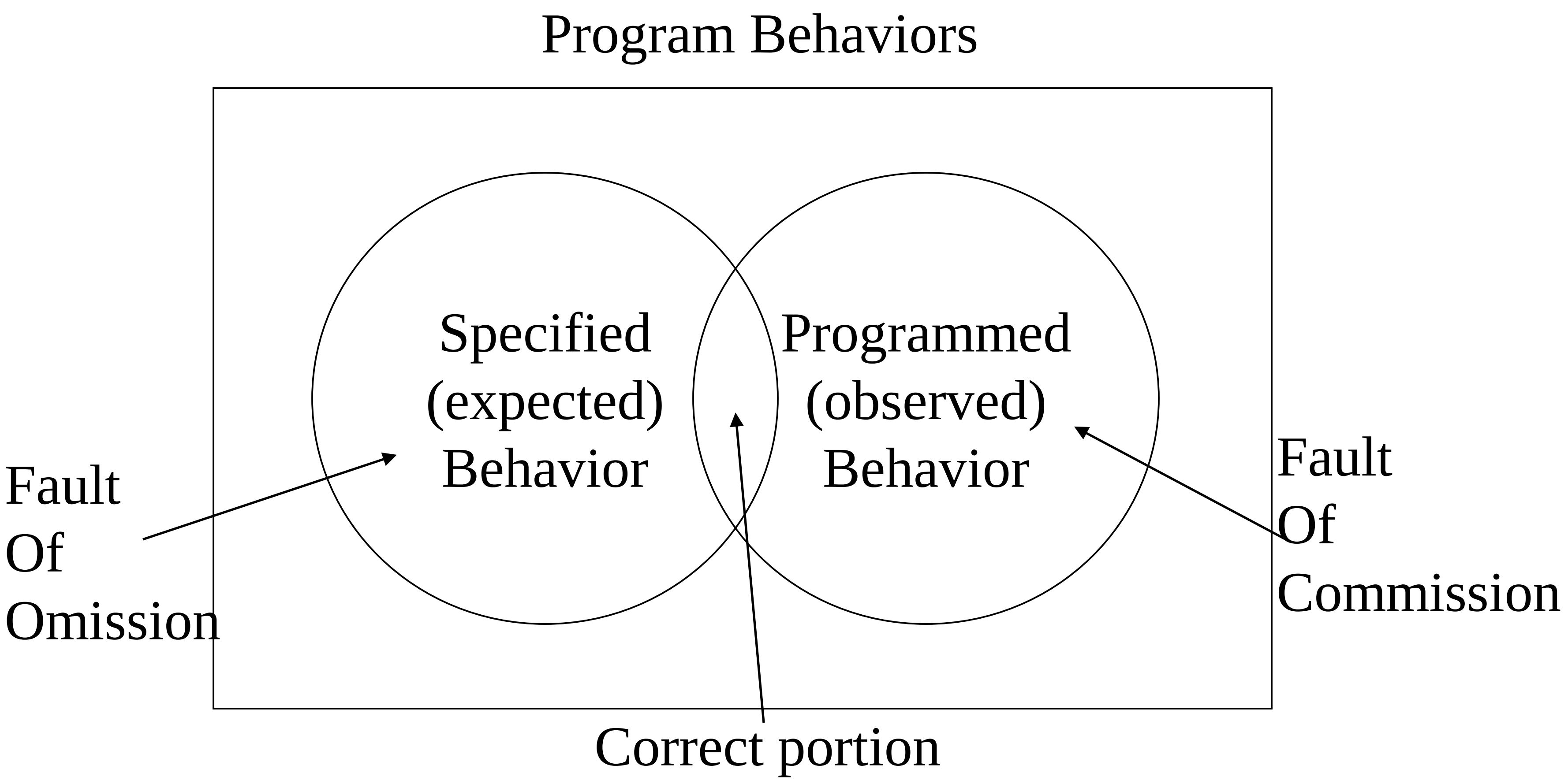
Typical test case information

Test case ID
Purpose
Preconditions
Inputs
Expected outputs
Post conditions
Execution History

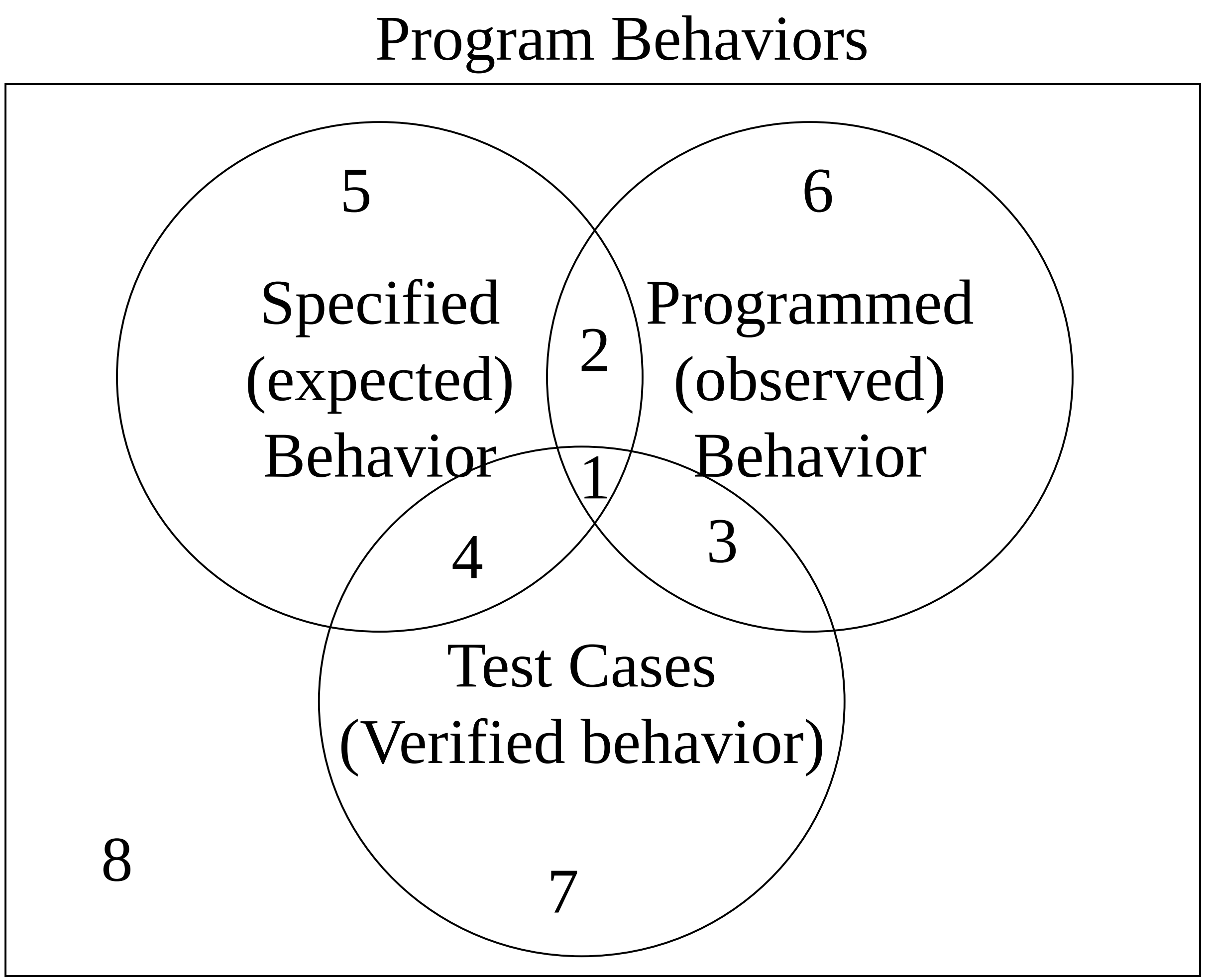
Insight from Venn diagram

- Two views
  - ✓ Structural view - what it is
  - ✓ Behavioral view - what it does – testing
- Difficulty of tester -Base document is only for developers

Relationship – program behaviors



Relationship – Testing wrt Behavior



Cont…

- 2, 5
  - Specified behavior that are not tested
- 1, 4
  - Specified behavior that are tested
- 2, 6
  - Programmed behavior that are not tested

Cont…

- 1, 3
  - Programmed behavior that are tested
- 3, 7
  - Test cases corresponding to unspecified behavior
- 4, 7
  - Test cases corresponding to un-programmed behaviors

Inferences

- If there are specified behaviors for which there are no test cases, the testing is incomplete
- If there are test cases that correspond to unspecified behaviors
  - Either such test cases are unwarranted
  - Specification is deficient

Test methodologies

- Functional (Black box) testing
- Structural (White box) testing

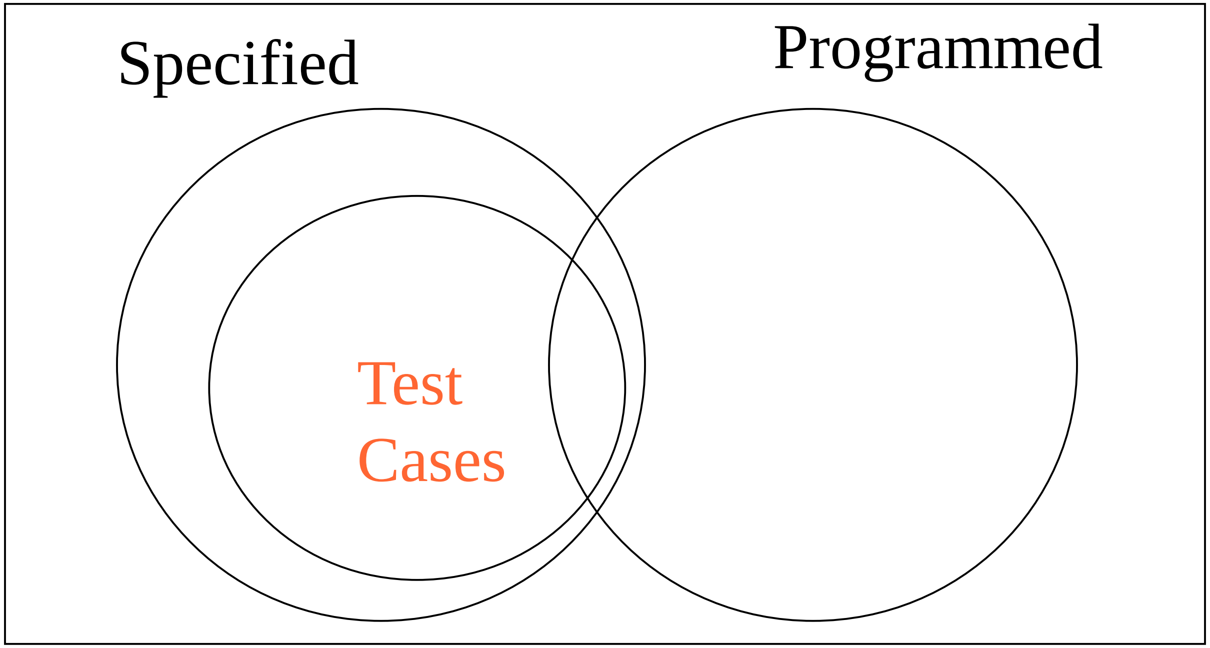
Functional Testing/Black box testing

- Program-a function that maps values from its input domain to values in its output range
- Content/implementation is not known
- Function is understood completely in terms of its inputs & outputs
- For test case identification only specification of the software is used

Advantages & Disadvantages of Functional Testing

- Advantages
  - ✓ Independent of software implementation
  - ✓ Test case development can occur in parallel
- Disadvantage
  - ✓ Redundancy among test cases.

Functional Test cases

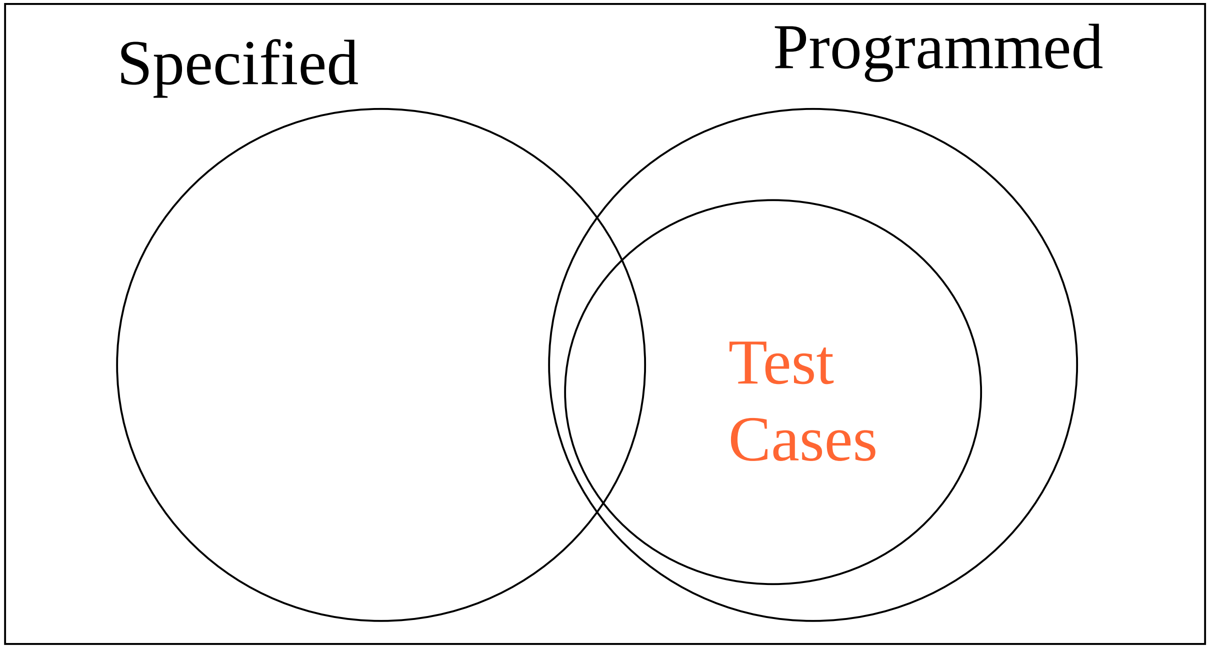


Functional methods are based on the specified behaviors only

Structural /white box /clear box testing

- Implementation is known and used to identify test cases
- Concept of linear graph theory is required to understand
- Test coverage metrics –provides way to state the extent to which the software item can be tested.

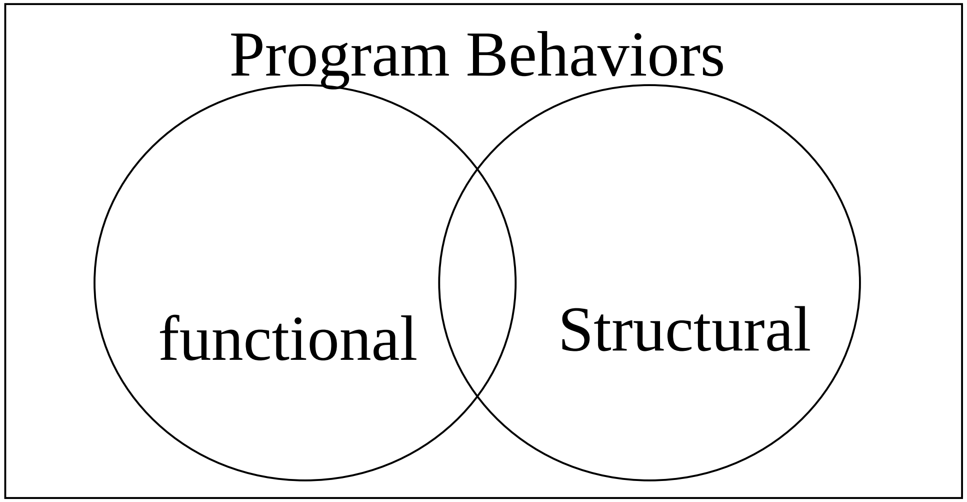
Structural Test cases



Structural methods are based on the programmed behaviors only

Functional verses structural

- Redundancy and gaps – problems of functional testing
- Functional test cases executed in combination with structural test coverage methods both problems can be recognized and solved.



Errors and fault taxonomies

- Process - how we do something
- Product - end result of a process
- Software quality assurance
  - ✓tries to improve product by improving process
  - ✓Concerned with reducing errors in development phases
  - ✓Testing concerned with discovering faults in a product-product oriented.

Classification of faults

- Based on Anomaly occurrence
  - ✓One time only
  - ✓Intermittent
  - ✓Recurring/repeatable

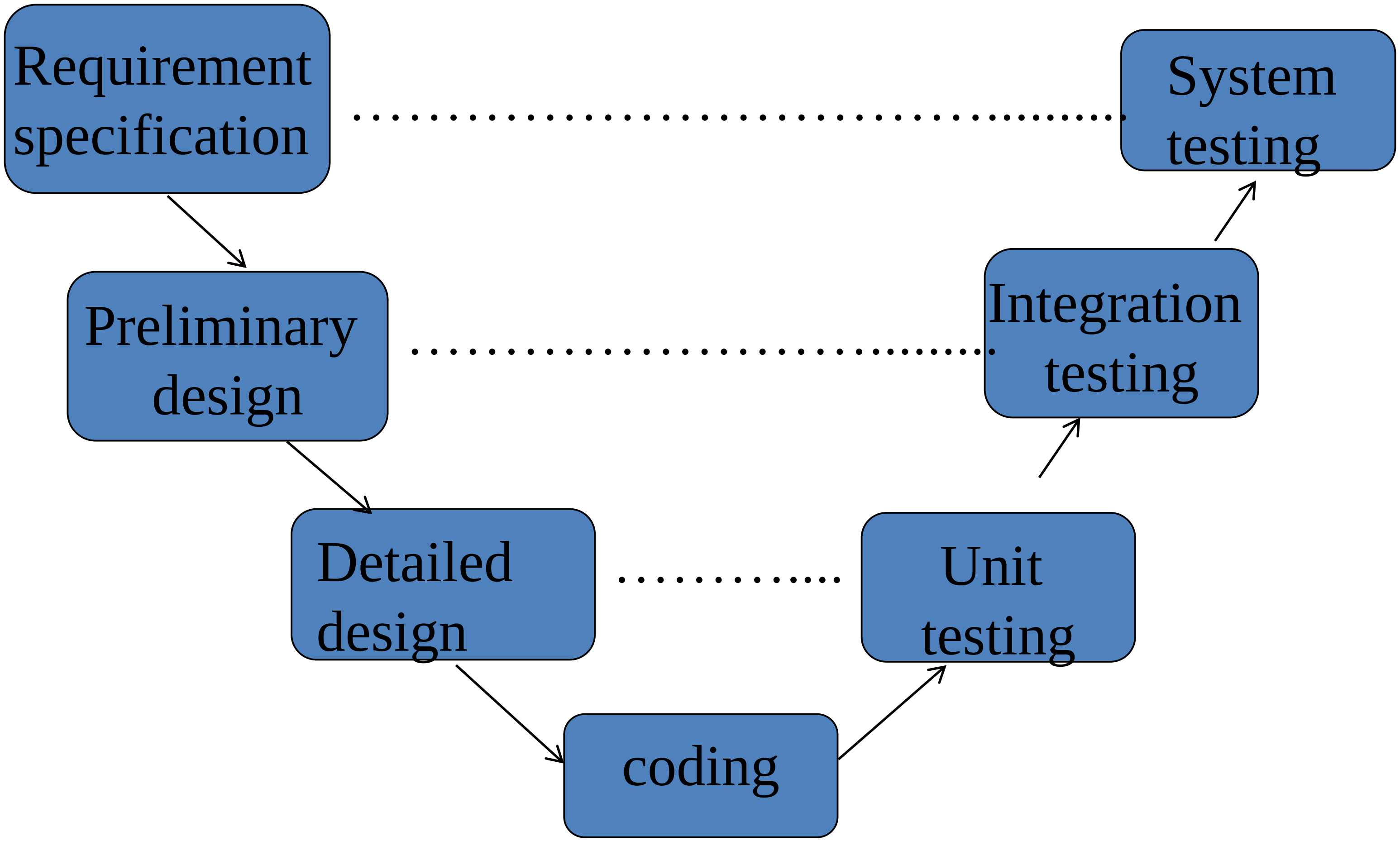
Based on severity

Mild	Misspelled word
Moderate	Misleading or redundant information
Disturbing	Some transactions not processed
Serious	Lose a transaction
Very serious	Incorrect transaction execution
Extreme	Frequent ”very serious” errors
Intolerable	Database corruption
Catastrophic	System shutdown

- Input / output faults
  - correct i/p not accepted
  - wrong format
  - wrong results
- Logic faults
  - missing condition
  - missing cases
  - Incorrect operand/operation
- Computational faults
  - incorrect algorithms
  - missing computations
  - Parenthesis error

- Interface faults
  - I/o timing
  - Incorrect i/p handling
  - Call to wrong procedure
- Data faults
  - Incorrect initialisation
  - Incorrect storage/access
  - Wrong flag/index value
  - Incorrect type

Levels of testing



Generalized pseudo code

- Provides “language neutral” way
- Levels of constructs
  - Program component
  - Unit component
    - traditional components → Procedure & functions
    - object oriented components → Class & object

Table 2.1 Generalized Pseudocode

Language Element	*Generalized Pseudocode Construct*
Comment	' <text>
Data structure declaration	Type <type name><list of field descriptions>End <type
Data declaration	Dim <variable> As <type>
Assignment statement	<variable> = <expression>
Input	Input (<variable list>)
Output	Output (<variable list>)
Condition	<expression> <relational operator> <expression>
Compound condition	<Condition> <logical connective> <Condition>
Sequence	Statements in sequential order
Simple selection	If <condition> Then <then clause>EndIf
Selection	If <condition>
Multiple selection	Case <variable> Of Case 1: <predicate> <Case clause> ... Case n: <predicate> <Case clause> EndCase
Counter-controlled repetition	For <counter> = <start> To <end>
Pretest repetition	While <condition> ... End While
Posttest repetition	Do ... until <condition>
Procedure definition (similarly for functions and o-o methods)	<procedure name>(Input: <list of variables>;Output: <l variables>)
Interunit communication	Call <procedure name> (<list of variables>; <list of variables>)
Class/object definition	<name> (<attribute list>; <method list>, <body>)End <
Interunit communication	msg <destination object name>.<method name> (<list of variables>)
Object creation	Instantiate <class name>.<object name> (list of attribute
Object destruction	Delete <class name>.<object name>
Program	Program <program name>

The Triangle Problem

Problem statement

Simple version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle.

The output of the program is the type of triangle determined by the three sides: Equilateral

Isosceles

Scalene

Not A Triangle.

The Triangle Problem

Improved version: "Simple version" plus better definition of inputs:

The integers a, b, and c must satisfy the following conditions:

- ✓ c1. $1 \leq a \leq 200$
- ✓ c2. $1 \leq b \leq 200$
- ✓ c3. $1 \leq c \leq 200$
- ✓ c4. $a < b + c$
- ✓ c5. $b < a + c$
- ✓ c6. $c < a + b$

The Triangle Problem

Final Version: "Improved version" plus better definition of outputs:

- ✓ If an input value fails any of conditions c1, c2, or c3, the program notes this with an output message "Value of b is not in the range of permitted values."

for example,

If values of a, b, and c satisfy conditions c1, c2, and c3, one of four mutually exclusive outputs is given:

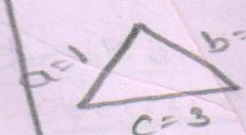
- ✓ If all three sides are equal, the program output is Equilateral.
- ✓ If exactly one pair of sides is equal, the program output is Isosceles.
- ✓ If no pair of sides is equal, the program output is Scalene.
- ✓ If any of conditions c4, c5, and c6 is not met, the program output is Not a Triangle.

Traditional Implementation

24

```
program triangle1 ' Fortran - like version
'
Dim a, b, c, match As INTEGER
'
Output ("Enter 3 integers which are sides of a triangle")
 Input (a, b, c)
 Output ("Side A is ", a)
 Output ("Side B is", b)
 Output ("Side C is", c)
 match = 0
 If a = b
    then match = match + 1
 End If
 If a = c
    Then match = match + 2
  End If
 If b = c
   Then match = match + 3.
  End If
   If match = 0
     Then If (a+b) <= c Then Output ("Not a triangle")
        Else If (b+c) <= a Then Output ("Not a triangle")
        Else If (a+c) <= b Then Output ("Not a Triangle"
          else Output ("Scalene")
                EndIf
           End If
End If
```

```
Else If match = 1
    Then If (a+2) <= b
        Then output ("NOT Triangle")
        Else output ("Isocelus")
    End if
Else If match = 2
    Then If (a+c) <= b Then output ("NOT Triangle")
                       Else output (" ISOSceus")
        Else
        End If
Else If match = 3
    Then If (b+c) <= b Then output (" NOT A Triangle")
                       Else output (" ISOsecus").
        End If
Else output ("Equilateral")
End If
End If
End If
End If
End Triangle.
```



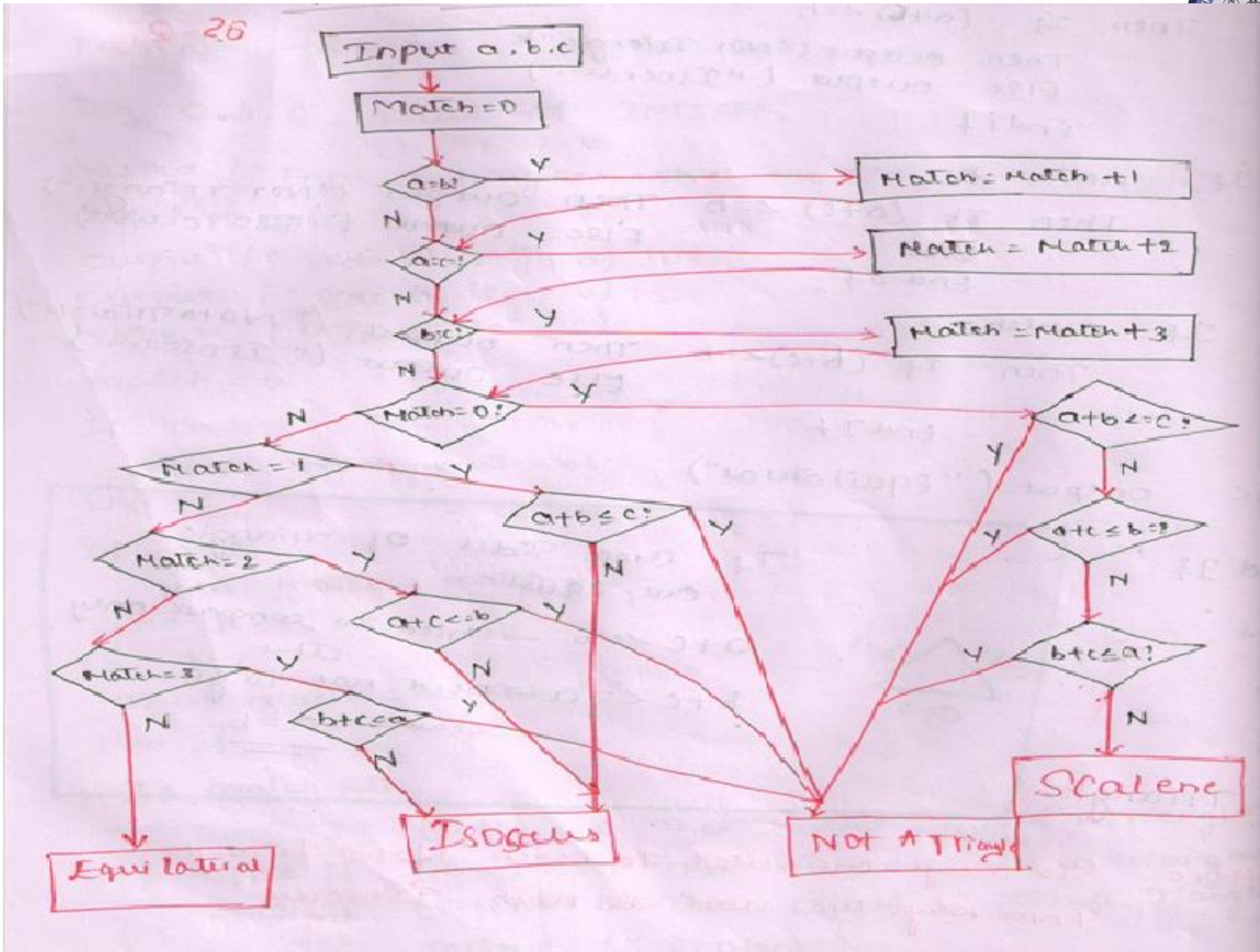
If two sides of triangle are equal

a+c <= b → need to compare only this

b+c <= a → need not to go because a = b

Here Six ways are used to reach NOT A Triangle

Three ways are used to reach ISOSceus

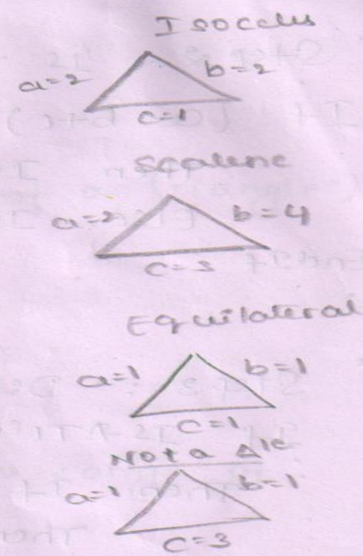


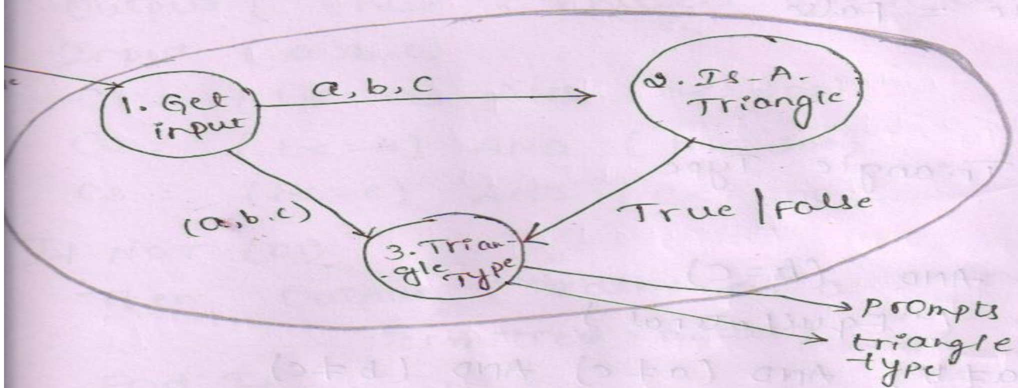
26
Input a.b.c
Match=0
a=b?
Match=Match+1
a=c?
Match=Match+2
b=c?
Match=Match+3
Match=0?
a+b<=c?
a+c<=b?
b+c<=a?
Match=1
a+b<=c?
Match=2
a+c<=b
Match=3
b+c<=a
Equilateral
Isosceles
Not A Triangle
Scalene
Y
N

Structured Implementation

Data flow diagram for a Structured triangle program implementation.

27





Program triangle 2 'Structured programming version of Simpler Specification'

Dim a, b, c As Integer
Dim IsATriangle As Boolean

Step 1 : Get Input
output ("Enter 3 integers which are sides of a triangle")

```
Output ("Side A is ", a)
Output ("Side B is ", b)
Output ("Side C is ", c)
'
' Step 2 : Is A Triangle ?
If (a< b+c) AND (b< a+c) AND (c< a+b)
    Then IsATriangle = True
    Else IsATriangle = False
EndIf
'
' Step 3 : Determine Triangle Type
If IsATriangle
    Then If (a= b) AND (b= c)
        Then Output ("Equilateral")
        Else If (a≠b) AND (a≠c) AND (b≠c)
            Then Output ("Scalene")
            Else Output ("Isosceles")
        EndIf
    EndIf
    Else Output ("Not a Triangle")
End If
end Triangle 2
```

23

Program Triangle3 'Structured programming version of improved specification'


```
Dim a, b, c As Integer
Dim C1, C2, C3, IsATriangle As Boolean

Step1 : Get Input
Do
    Output ("Enter 3 integers which are Sides of a triangle")
    Input (a, b, c)
    C1 = (1 <= a) AND (a <= 200)
    C2 = (1 <= b) AND (b <= 200)
    C3 = (1 <= c) AND (c <= 200)
  If NOT (C1)
   Then  Output ("value of a is not in the range of
              permitted values")
   End If
   If NOT (C2)
   Then  output ("value of b is not in the range of
              permitted values")
    End If
    If NOT (C3)
    Then  output ("value of c is not in the range of
              permitted values")
    End If
Untill  C1 AND C2 AND C3  → Till All 3 condition
```

```
Output ("Side A is", a)
Output ("Side B is", b)
Output ("Side C is", c)

Step 2: Is A Triangle:
  If (a <(b+c)) AND (b < (a+c)) AND (c < (a+b))
      Then IsATriangle = True
       Else IsATriangle = False
    EndIf

Step 3: Determine Triangle-Type
    If IsATriangle
      Then If (a=b) AND (b=c)
            Then output ("Equilateral")
            Else If (a≠b) AND (a≠c) AND (b≠c)
                  Then output ("scalene")
                  Else output ("Isoscles")
                EndIf
           EndIf
     Else output ("Not a triangle")
   EndIf
End triangle 3.
```

Complexity in the triangle problem is due to relationships b/w inputs & correct outputs

The NextDate function

- NextDate is a function of three variables:
  - ✓ month
  - ✓ Date
  - ✓ Year
- ➢ returns the date of the day after the input date
- The month, date, and year variables have integer values subject to these conditions:
  - ✓ c1. $1 \le \text{month} \le 12$
  - ✓ c2. $1 \le \text{day} \le 31$
  - ✓ c3. $1812 \le \text{year} \le 2012$

The NextDate function

- If any of conditions c1, c2, or c3 fails → outputs → variable has an out-of-range value
- If i/p value is invalid → outputs → invalid input date
- Two source of complexity
  - ✓ Complexity of input domain
  - ✓ Rule that determines when a year is a leap year
- Leap year problem is solved by Gregorian calendar

" Year is leap year if it is divisible by 4 i.e. only for non-century year "

Century year is a leap year if it is divisible by 400. Thus 1600, 2000, 2004 and 2008 are leap years, but 1700, 1900 and 2100 are not"

Implementation

```
Program NextDate1 'Simple version
'
Dim tomorrowDay, tomorrowMonth, tomorrowYear As Integer
Dim day, month, year As Integer
'
Output (" Enter today's date in the form MM DD YYYY")
Input (month, day, year)
Case month Of
Case 1: month Is 1, 3, 5, 7, 8 or 10: '31 day month (except Dec.)
    If day < 31
      Then tomorrowDay = day+1
      Else
        tomorrow Day=1
        tomorrowMonth = month+1
    EndIf
-------------------------------------------------------------
Case 2: month Is 4, 6, 9, or 11 '30 day month
    If day <30
      Then tomorrowDay = day +1
      Else
        tomorrowDay =1
        tomorrowMonth = month+1
    EndIf
```

Case 3: month is 12: ' December

```
If day < 31
 Then tomorrowDay = day+1
 Else
   tomorrow Day = 1
   tomorrowMonth = 1
   If year = 2012
     Then O/P ("2012 is over")
     Else tomorrow. Year = Year+1
   EndIf
EndIf
```

Case 4: month is 2: ' February

```
If day < 28
 Then TomorrowDay = day+1
 Else
  If day = 28
   Then
    If ((year is a leap year)
     Then TomorrowDay = 29 'leap year
     Else 'not a leap year
          tomorrow day = 1
          tomorrowmonth = 3
    EndIf
 Else If day = 29
      then Tomorrow day = 1
       tomorrow month = 3
Else Output ("Cannot have Feb.", day)
```

```
End if
End if
End if
End case
output ("Tomorrow's date is", tomorrowMonth, tomorrow Day,
        tomorrow year)
'
END Next Date.
```

Program Next Date2 Improved Version

```
'
Dim tomorrowDay, Tomorrow Month, TomorrowYear As Integer
Dim day, month, year As Integer
Dim C1, C2, C3 AS Boolean
'
Do
   Output (" Enter Today's date in the form MM DD YYYY")
   Input (month, day, year)
    C1 = (1<=day) AND (day <=31)
    C2 = (1<=month) AND (month <=12)
    C3 = (1812 <= year) AND (Year <= 2012)
  If NOT (C1)
    Then   output (" value not in range)
    End If
  If NOT (C2)
    Then   output (" value not in range)
    End If
```

35

```
If NOT (C3)
Then output ("value of year not in range)
End if
Until C1 AND C2 AND C3

Case month of
Case 1 : month Is 1, 3. 5. 7. 8. Or 10 : '31 day months (except Dec)
  If day < 31
    then Tomorrow day = day +1
  else
        tomorrow day = 1
        tomorrow month = month + 1
  Endif
Case 2 : month Is 4, 6, 9, or 11 '30 day months
  If day < 30
    then tomorrowDay = day +1
  Else
    If day = 30
      Then tomorrowDay = 1
           tomorrow month = month + 1
      else output ("Invalid Input Date")
    Endif
  Endif
```

Case 3: month is 12 : December

If ~~36~~ day < 31
Then tomorrowDay = day + 1
Else
tomorrowDay = 1
tomorrow Month = 1
If year = 2012
then output ("Invalid Input Date")
Else tomorrow . year = year + 1
Endif
Endif

Case 4: month is 2 : 'February
If day < 28
Then tomorrowDay = day + 1
Else
If day = 28
then
if (year is a leap year)
then tomorrowDay = 29 'leap year
else 'not a leap year
tomorrowday = 1
tomorrow month = ~~month + 1~~ 3
endif
else
if day = 29
then if (year is a leap year)
then tomorrowDay = 1
tomorrow Month = 3

```
else
    if day > 29
        Then output ("Invalid input date")
      endif
   Endif
 Endif
Endif
End case
Output ("Tomorrow's date is", tomorrowMonth,
     tomorrowDay, TomorrowYear)

End NextDate2
```

11

The Commission problem

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri.

- ✓ **Locks cost $45**
- ✓ **stocks cost $30**
- ✓ **barrels cost $25**.

The salesperson had to sell at least one complete rifle per month, and production limits were such that the most the salesperson could sell in a month was

- ✓ **70 locks**
- ✓ **80 stocks**
- ✓ **90 barrels.**

The Commission problem

After each town visit, the salesperson sent a telegram to the Missouri gunsmith with the number of locks, stocks, and barrels sold in that town. At the end of a month, the salesperson sent a very short telegram showing –1 locks sold.

salesperson's commission is computed as follows:

- ✓ **10% on sales up to (and including) $1000**
- ✓ **15% on the next $800**
- ✓ **20% on any sales in excess of $1800**.

The commission program produced a monthly sales report that gave the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales, and, finally, the commission.

The Commission problem

- This problem separates into three distinct pieces
  - ✓Input data portion-deals with input data validation
  - ✓Sales calculation
  - ✓Commission calculation portion

Discussion

Problem separates into three distinct pieces

i) input data portion - deal with input data validation

ii) Sales Calculation

iii) Commission calculation portion.

Implementation

```
Program Commission (INPUT, OUTPUT)
'
Dim locks, Stocks, barrels As Integer
Dim lockprice, Stock price, barrel price As Real
Dim totalLocks, total Stocks, Total Barrels As Integer
Dim lock sales, stocksales, barrel sales As Real
Dim sales, commission : REAL
'
   Lock price = 45.0
   Stock price = 30.0
   barrelprice = 25.0
   total lock = totalStock = Total Barrel = 0;
'
 Input (locks)
While NOT (locks = -1)      ' input device uses -1 to indicate
                              end of data'
  Input (Stocks, barrels)
   Total locks = total Locks + locks
   Total Stocks = total Stocks + Stocks
   Total Barrels = total Barrels + barrels
     Input (locks)
EndWhile
'
```

```
Output ("Locks sold : ", totalLocks)
Output ("Stocks Sold:", totalStocks)
Output ("Barrels sold:", totalBarrels)
```

13

Sales Calculation

```
lockSales = lockprice * totalLocks
stockSales = stockprice * Totalstocks
barrel Sales = barrelprice * totalBarrels
Sales = locksales + stocksales + barrelSales
output ("Total Sales : ", Sales)
```

Commission calculation

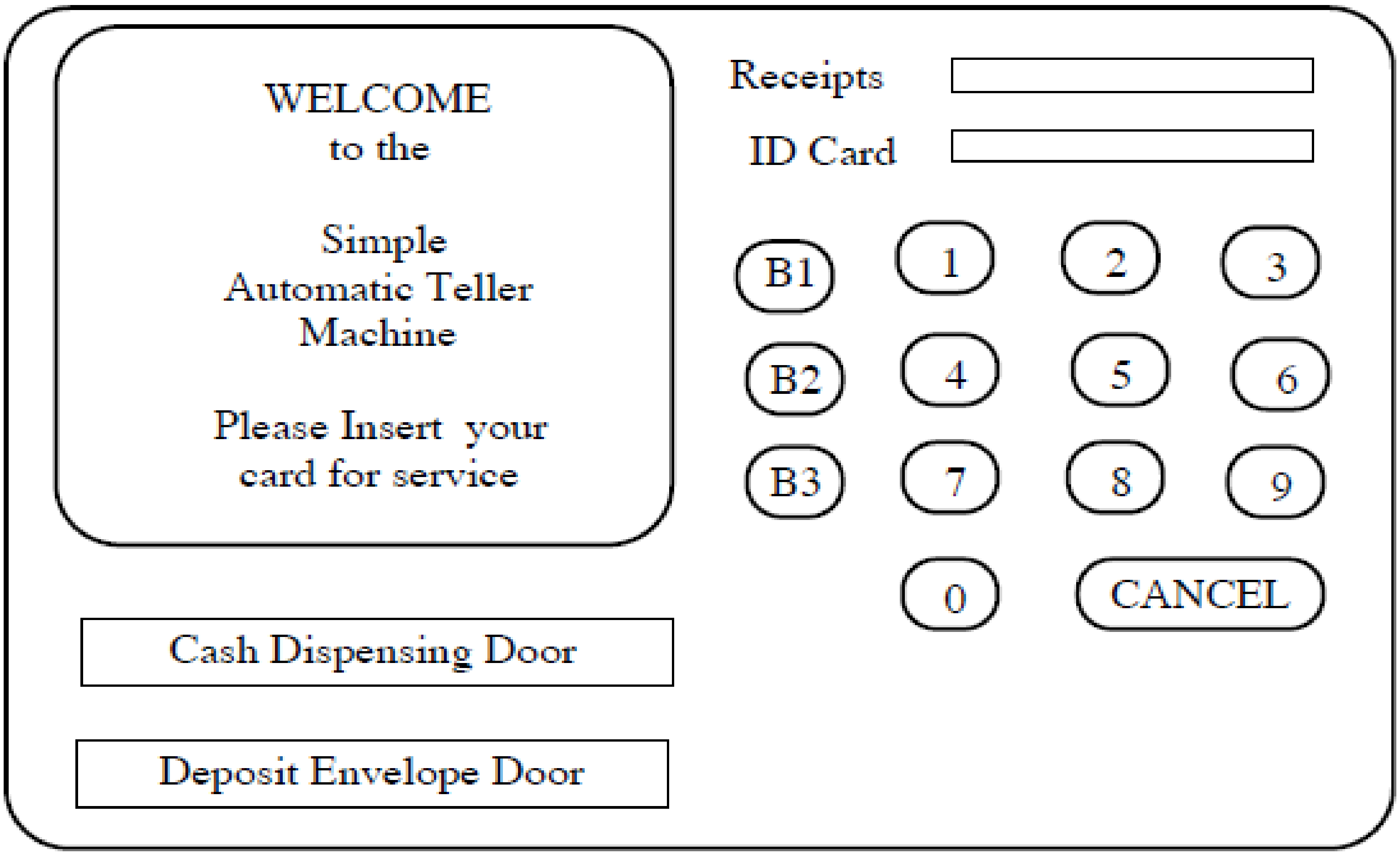
```
if (Sales > 1800.0)
  Then
      Commission = 0.10 * 1000.0
      Commission = Commission + 0.15 * 800.0
      Commission = Commission + 0.20 * (Sales - 1800.0)
   Else If (Sales > 1000.0)
     Then
         Commission = 0.10 * 1000.0
         Commission = Commission + 0.15 * (Sales - 1000.0)
     Else commission = 0.10 * Sales
     EndIf
EndIf
output ("Commission is $", Commission)

End Commission.
```

The SATM System

Problem statement

- The SATM(Simplified Automated Teller Machine) system communicates with bank customers via the 15 screens.
- Customers can select any of three transaction types
  - ✓Deposits
  - ✓Withdrawals
  - ✓Balance enquires
- Transactions can be done on two types of account
  - ✓Checking
  - ✓savings

The SATM System



Screen 1

Welcome.

Please insert your ATM card for service

Screen 2

Enter your Personal Identification Number

__ __ __ __

Press Cancel if Error

Screen 3

Your Personal Identification Number is incorrect. Please try again.

Screen 4

Invalid identification. Your card will be retained. Please call the bank.

Screen 5

Select transaction type:
balance
deposit
withdrawal

Press Cancel if Error

Screen 6

Select account type:

checking
savings

Press Cancel if Error

Screen 7

Enter amount. Withdrawals must be in increments of $10

__ __ __ __ . __ __

Press Cancel if Error

Screen 8

Insufficient funds. Please enter a new amount.

__ __ __ __ . __ __

Press Cancel if Error

Screen 9

Machine cannot dispense that amount.

Please try again.

Screen 10

Temporarily unable to process withdrawals. Another transaction?
yes
no

Screen 11

Your balance is being updated. Please take cash from dispenser.

Screen 12

Temporarily unable to process deposits. Another transaction?
yes
no

Screen 13

Please put envelope into deposit slot. Your balance will be updated.

Press Cancel if Error.

Screen 14

Your new balance is printed on your receipt. Another transaction?
yes
no

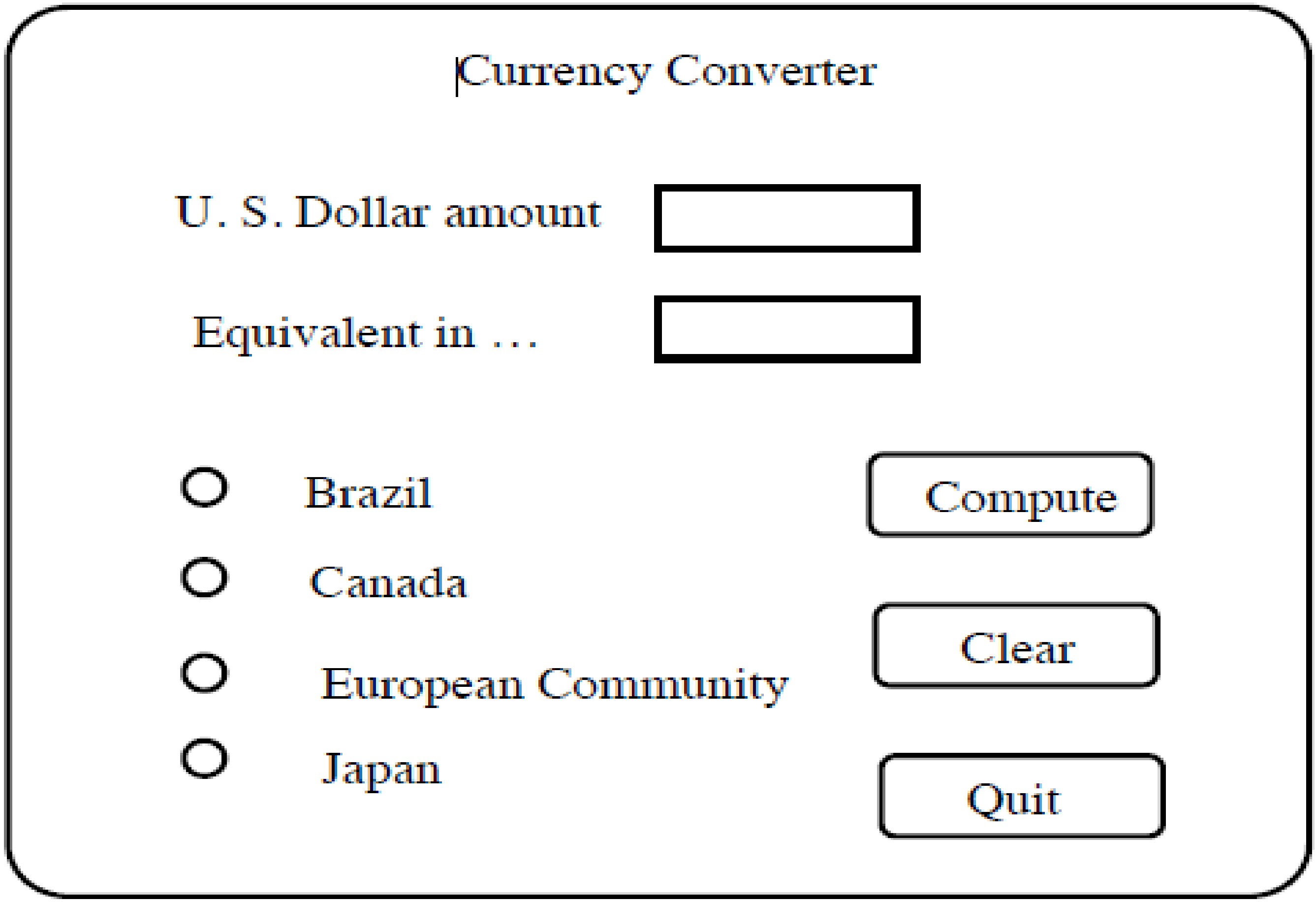
Screen 15

Please take your receipt and ATM card. Thank you.

The Currency Converter

- Event driven program
- Code associated with a graphical user interface(GUI)
- Works on the basis of completing label
- Users can click on
    - » Compute button
    - » Clear button
    - » Quit button

The Currency Converter



Saturn windshield wiper controller

- Controlled by lever with a dial
- Leaver positions
  - ✓OFF
  - ✓INT(intermittent)
  - ✓LOW
  - ✓HIGH
- Dial positions (1,2,3) indicates three intermittent speeds & is relevant only when lever is in INT position.

Saturn windshield wiper controller

- Decision table showing windshield wiper speeds(in wipes per minute) for the lever & dial position

Lever	OFF	INT	INT	INT	LOW	HIGH
Dial	n/a	1	2	3	n/a	n/a
Wiper	0	4	6	12	30	60

Wiper speeds in wipes per minute

Thank you

My Details

Dr. Manjunath T. N.
Professor
Dept.of.ISE
BMSIT, Bengaluru
Email: manju.tn@bmsit.in
manju.tn@gmail.com
Mobile:+91-9900130748

Software Testing

Module-2 : BVA,ECP & DTM

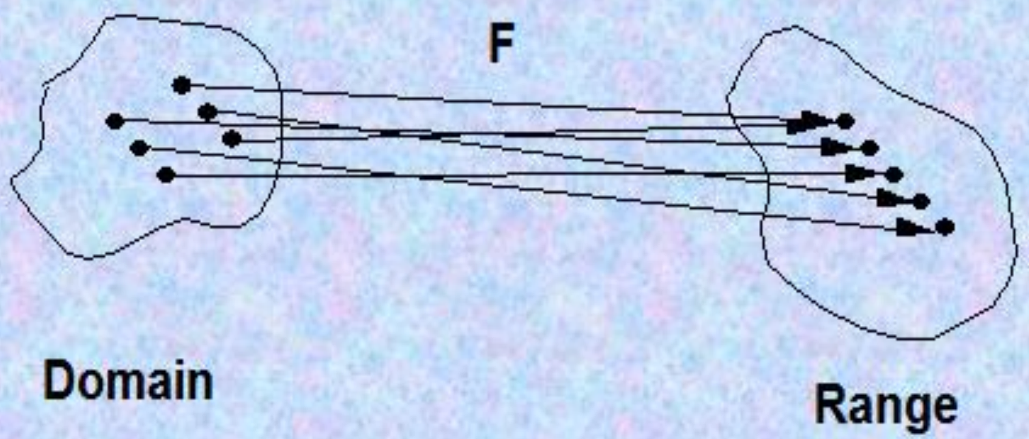
By,

Dr. Manjunath T. N.
Professor
Dept. of Information Science & Engg.
BMS Institute of Technology, Bengaluru.

Functional Testing

Ultimately, any program can be viewed as a mapping from its Input Domain to its Output Range:

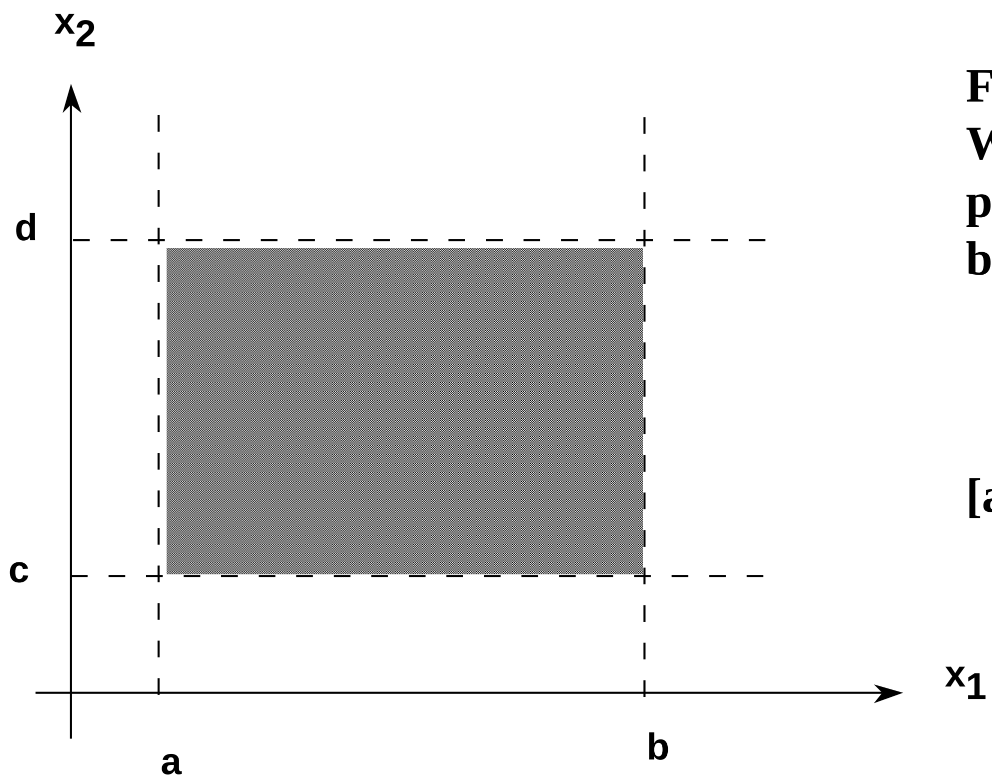
Output = F (Input)



Functional testing uses information about functional mappings to identify test cases:

Boundary value Analysis

Input Domain of $F(x_1, x_2)$



F –-function of two variables X1 & X2
When function is implemented as a program both X1 & X2 have some boundaries

$$a \le x1 \le b$$

$$c \le x2 \le d$$

[a,b] and [c,d] -range of x1 & x2

Boundary value Analysis

- It is a software testing technique in which tests are designed to include representative of boundary values
- Used to identify errors at boundaries rather than finding those exits in center of input domain
- Range checking—focuses on the boundary of the input space to identify test cases

Boundary value Analysis

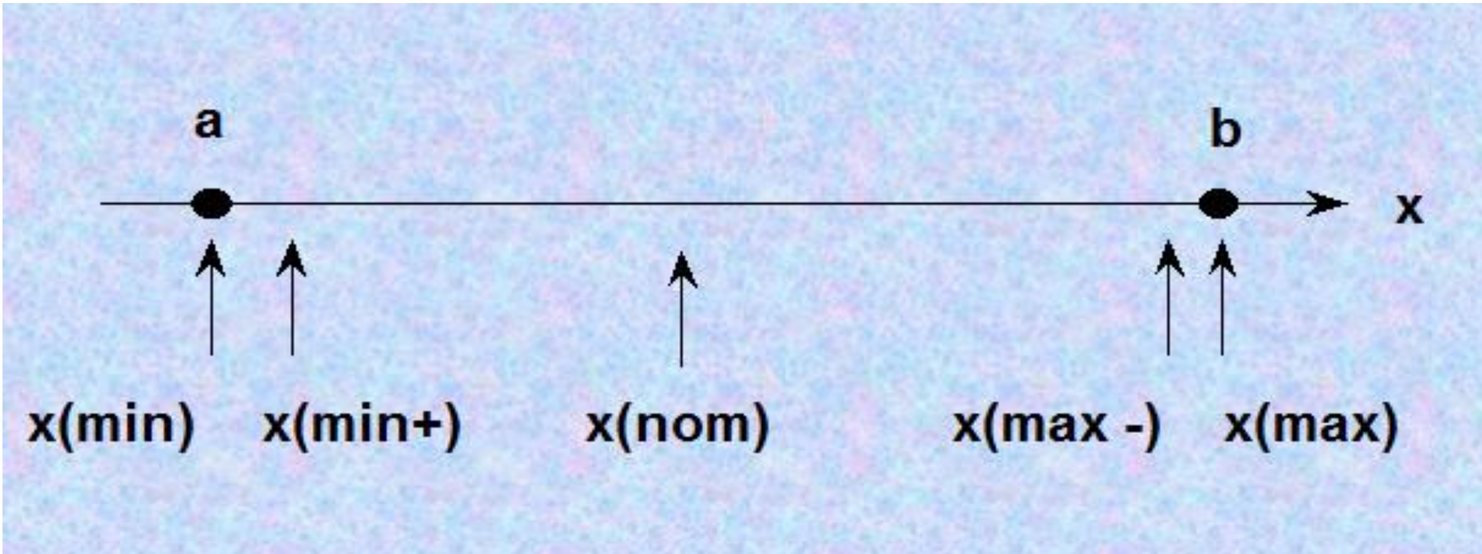
- In general application of Boundary Value Analysis can be done in a uniform manner
- The basic form of implementation is to maintain all but one of the variables at their nominal(normal or average) values and allowing the remaining variable to take on its extreme values

Boundary value analysis (cont…)

The basic idea is to use input variable values at their

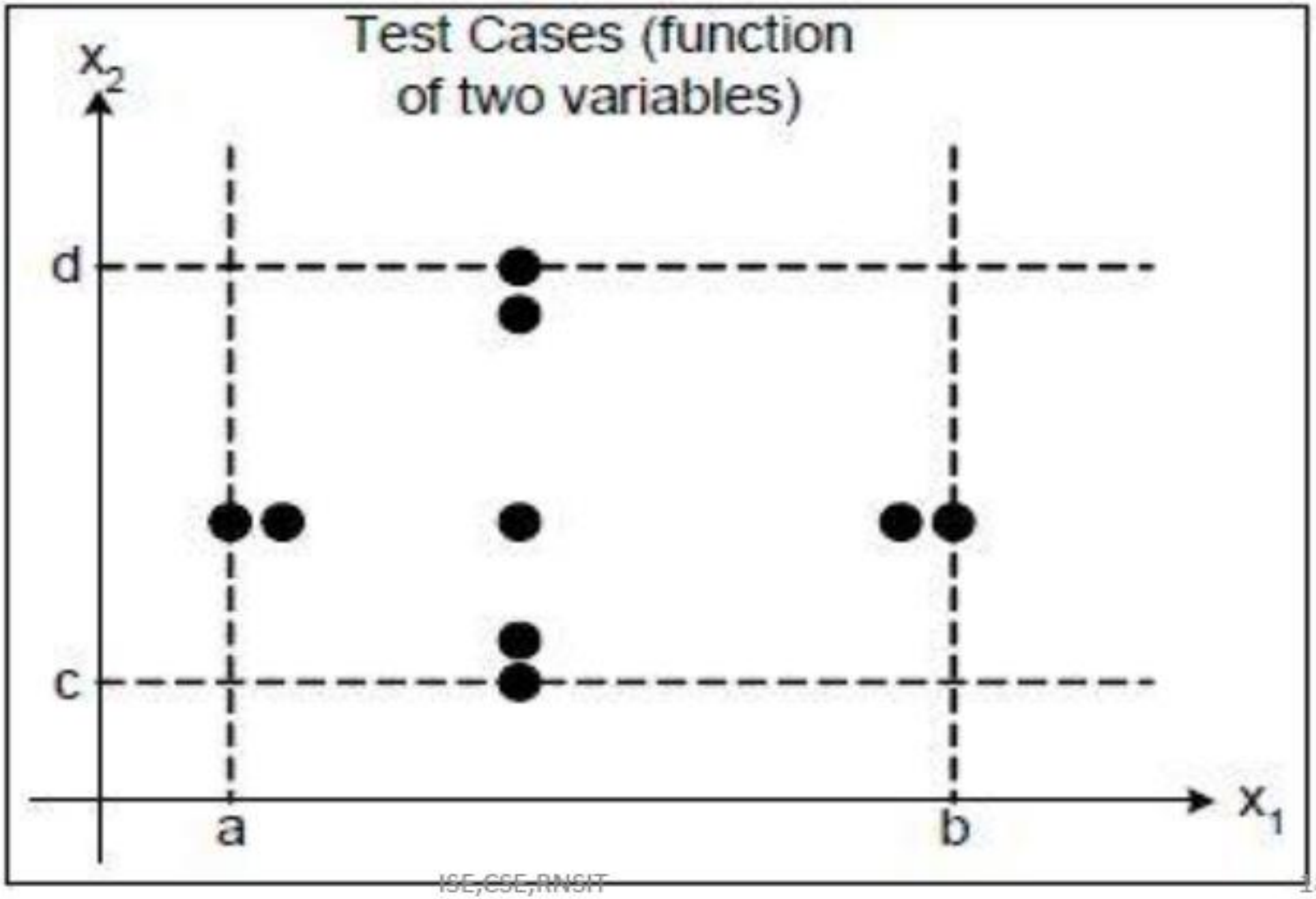
The values used to test the extremities are	
Min	Minimal
Min+	Just above Minimal
Nom	Average/Nominal/normal
Max-	Just below Maximum
Max	Maximum

Input Boundary value testing

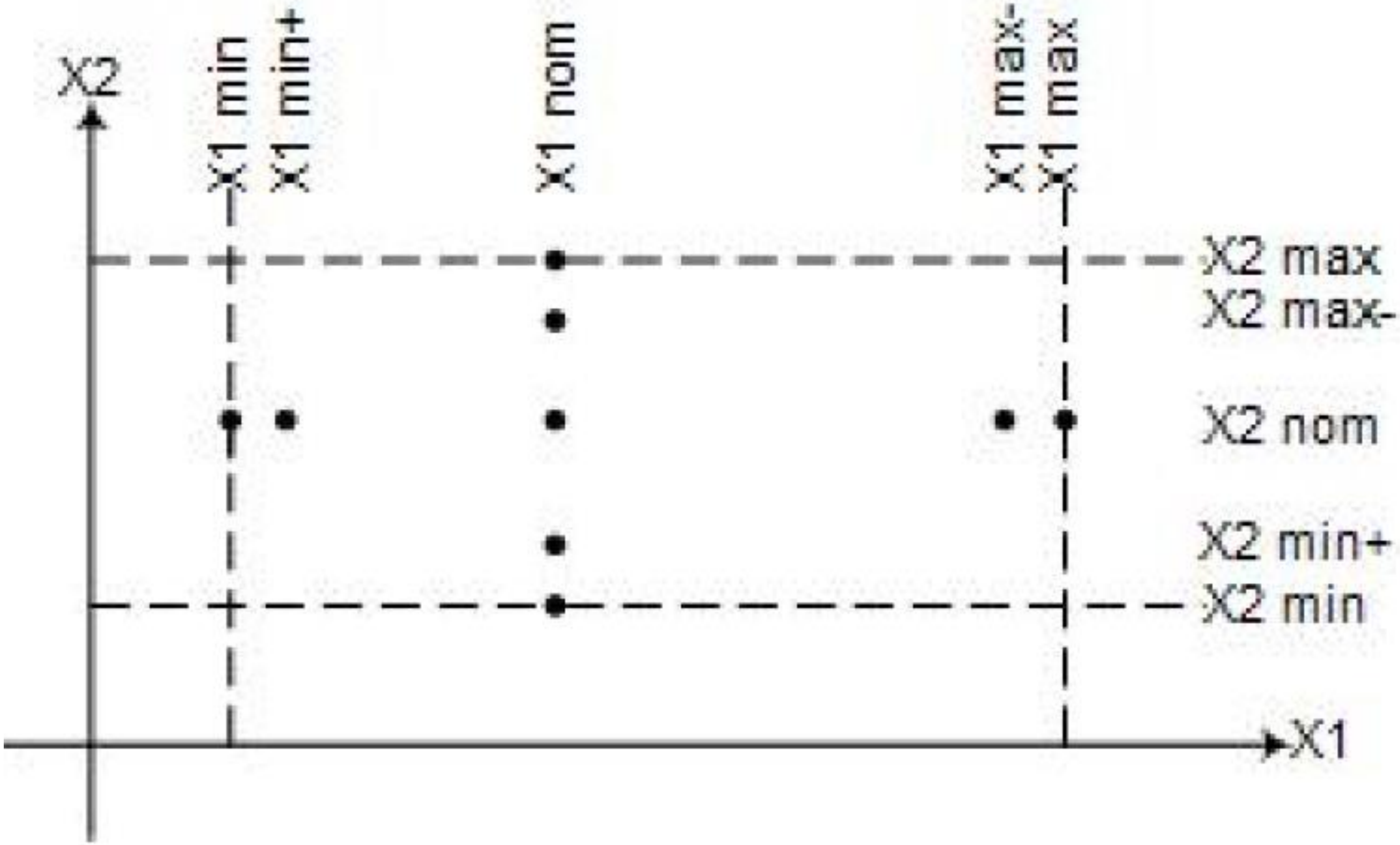


Experience shows that errors occur more frequently for extreme values of a variable.

Input Boundary value testing



Input Boundary value testing



Boundary value testing

Based on critical assumptions known as single fault assumption in reliability theory

- failures are only rarely the result of the simultaneous occurrence of two (or more) faults.
- Thus boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values & letting that variable assume its extreme values.

Generalizing Boundary value Analysis

- The basic boundary value analysis technique can be generalized in two ways:
    - ✓ Number of variables
    - ✓ Kinds of ranges
- Generalizing number of variables is easy
- Generalizing ranges depends on the nature of the variables themselves.

Boundary value analysis yields 4n+1 unique test cases

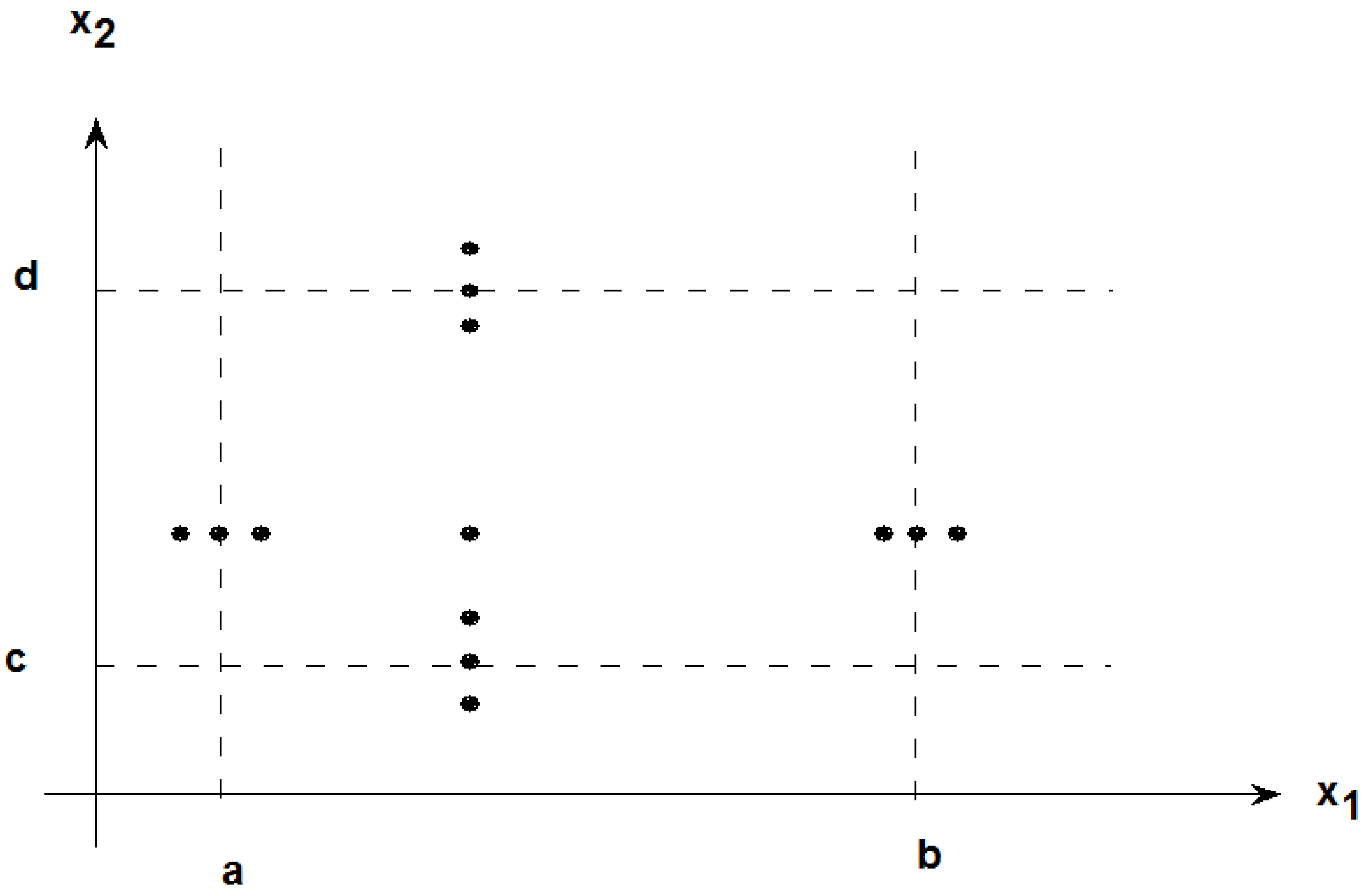
Limitations of boundary value analysis

- Works when the program to be tested is a function of several independent variable that represent bounded physical quantities.
- Boundary value analysis test cases are rudimentary
- Physical quantity criterion

Robustness testing

- Extension of boundary value analysis
- Shows what happens when extreme are exceeded with
    - ✓Value slightly greater than the maximum(max+)
    - ✓Value slightly less than the minimum(min-)

Robustness testing



Robustness testing

Forces attention on exception handling

Worst case testing

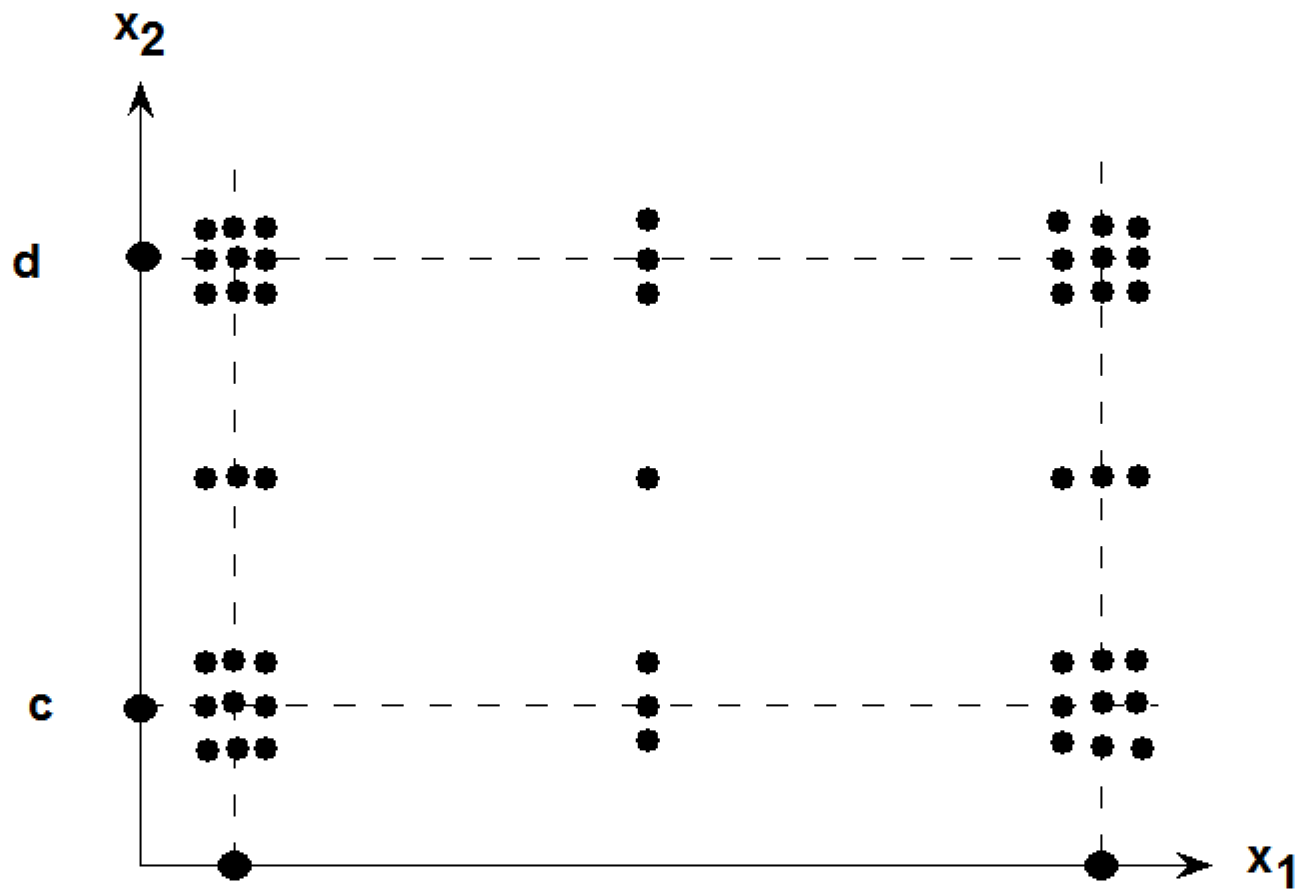
- Depends on single fault assumption of reliability theory

What happens when more than one variable has an extreme value –**WORST CASE ANALYSIS**

Generation of worst-case test cases

- Start with 5 element set {min,min+,nom,max-,max}
- Take Cartesian product of sets

Generation of worst-case test cases

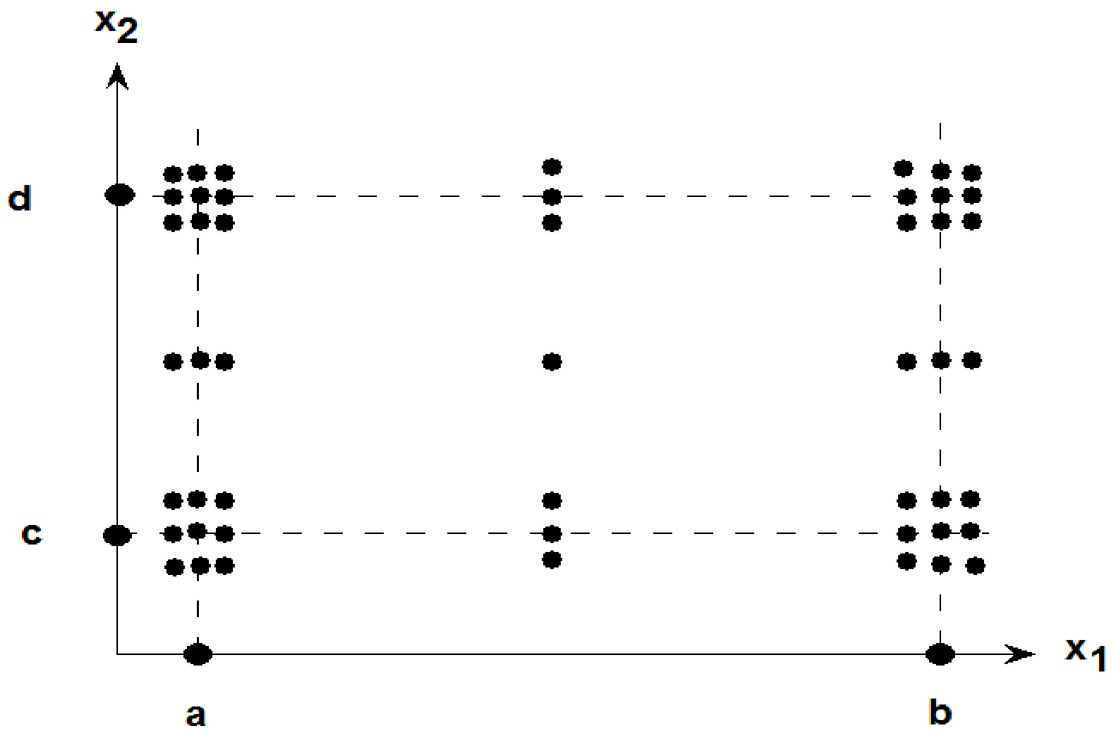


Relationship b/w boundary value & worst-case analysis

- boundary value analysis test cases are proper subset of worst-case test cases
- Effort is more– worst case testing for a function of n variables generates 5^n test cases as opposed to 4n+1 test cases for boundary value analysis.

Robustness worst case testing

- Involves the Cartesian product of seven elements sets results in 7^n test cases



Special value testing

- Adhoc testing
- Most widely practiced form of function testing
- Most intuitive and least uniform
- Occurs when a tester uses
- domain knowledge
- Experience with similar programs
- Information about ‘”soft spots” to device test cases
- Best engineering technique is used than guidelines
- Depends on ability of the tester

EXAMPLES

amples

Triangle Problem Boundary Value Analysis Test Cases.

a	b	c	Expected o/p	Reasons
100	100	1	Isosceles	100, 100 → 2 sides are equal
100	100	2	Isosceles	— " —
100	100	100	Equilateral	all sides equal
100	100	199	Isosceles	100, 100 → 2 sides are equal
100	100	200	Not a Δ^le	100+100 ≤ 200
100	1	100	Isosceles	
100	2	100	Isosceles	
100	100	100	Equilateral	all sides equal
100	199	100	Isosceles	
100	200	100	Not a Δ^le	100+100 ≤ 200
1	100	100	Isosceles	
2	100	100	Isosceles	
100	100	100	Equilateral	
199	100	100	Isosceles	
200	100	100	Not a Δ^le	100 + 100 ≤ 200

EXAMPLES

Triangle problem worst - case Test cases 26

↳ Shows the worst cases test cases in just "one corner" of the input space cube. (In Spread sheet)

Triangle

Case	a	b	c	Expected o/p	
1	1	1	1	Equilateral	
2	1	1	2	Not a Δle	1+1 ≤ 2.
3	1	1	100	Not a Δle	1+1 ≤ 100
4	1	1	199	Not a Δle	
5	1	1	200	Not a Δle	
6	1	2	1	Not a Δle	
7	1	2	2	Isosceles	
8	1	2	100	Not a Δle	1+2 ≤ 100
9	1	2	199	Not a Δle	
10	1	2	200	Not a Δle	
11	1	100	1	Not a Δle	
12	1	100	2	Not a Δle	

EXAMPLES

Test Cases for Next Date Function

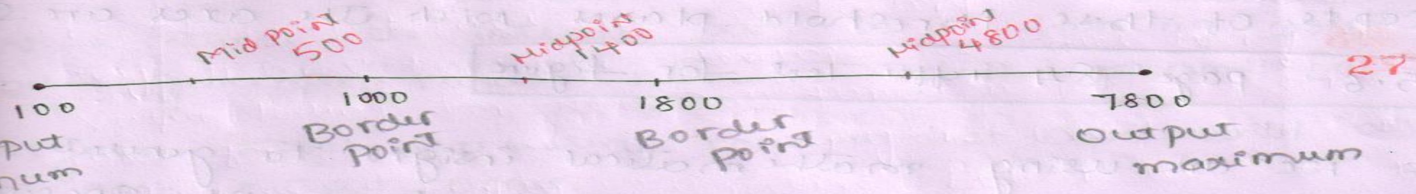
→ Contains the worst case test cases for the Next Date function Only one corner of input space cube (B) is shown:

Case	month	Day	year	Expected output
1	1	1	1812	Jan, 2, 1812
2	1	1	1813	Jan, 2, 1813
3	1	1	1912	Jan, 2, 1913
4	1	2	1812	Jan 3, 1812
5	1	31	1812	feb 1, 1812

continue

EXAMPLES

...st cases for the Commission problem



...mmission Problem Output Boundary Value Analysis Test cases

...e	Locks	Stocks	Barrels	Sales	Commission	Comment
	1	1	1	100	10	Output minimum
	1	1	2	125	12.5	Output minimum +
	1	2	1	130	13	Output minimum +
	2	1	1	145	14.5	Output minimum +
	5	5	5	500	50	Mid point
6	10	10	9	975	97.5	Border point −
7	10	9	10	970	97	Border point −
8	9	10	10	955	95.5	Border point −
9	10	10	10	1000	100	Border point
10	10	10	11	1025	103.75	Border point +
11	10	11	10	1030	104.5	Border point +
12	11	10	10	1045	106.75	Border point +
13	14	14	14	1400	160	Mid point
14	18	18	17	1775	216.25	Border point −
15	18	17	18	1770	215.5	Border point −
16	17	18	18	1775	213.25	Border point −
17	18	18	18	1800	220	Border point
						Border point +

Random testing

Rather than always choose the min, min+, nom,max+, max values of a bounded variables use a random number generator to pick test case values.

How many Random test cases are sufficient ?

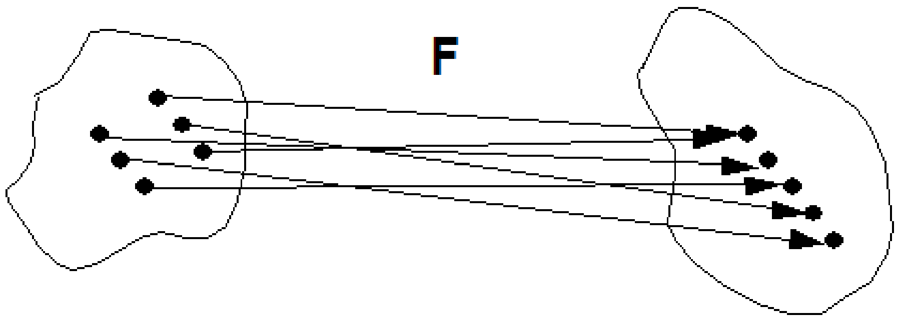
- Structural test average metrics gives answer
- X= Int ((b-a+1)* Rnd +a)
    - ✓fun Int--returns the integer part of a floating point number
    - ✓fun Rnd– generates random numbers in the interval[0,1]
- Program keeps generating random test cases until at least one of each output occur.

Random test cases for Triangle problem

Table 5.6 Random Test Cases for the Triangle Program

Test Cases	Nontriangles	Scalene	Isosceles	Equilateral
1289	663	593	32	1
15436	7696	7372	367	1
17091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
Percentage	49.83%	47.87%	2.29%	0.01%

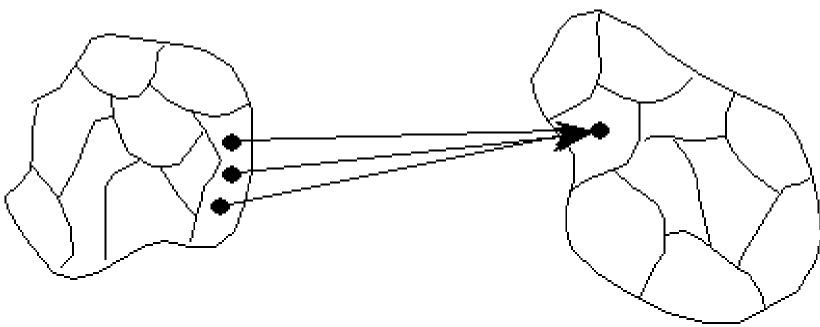
Equivalence Class Testing



Define relation R as follows:

for x, y in domain, xRy iff F(x) = F(y).

Equivalence relation



Test cases are formed by selecting one value from each equivalence class.

- reduces redundancy
- identifying the classes may be hard

Equivalence Class Testing

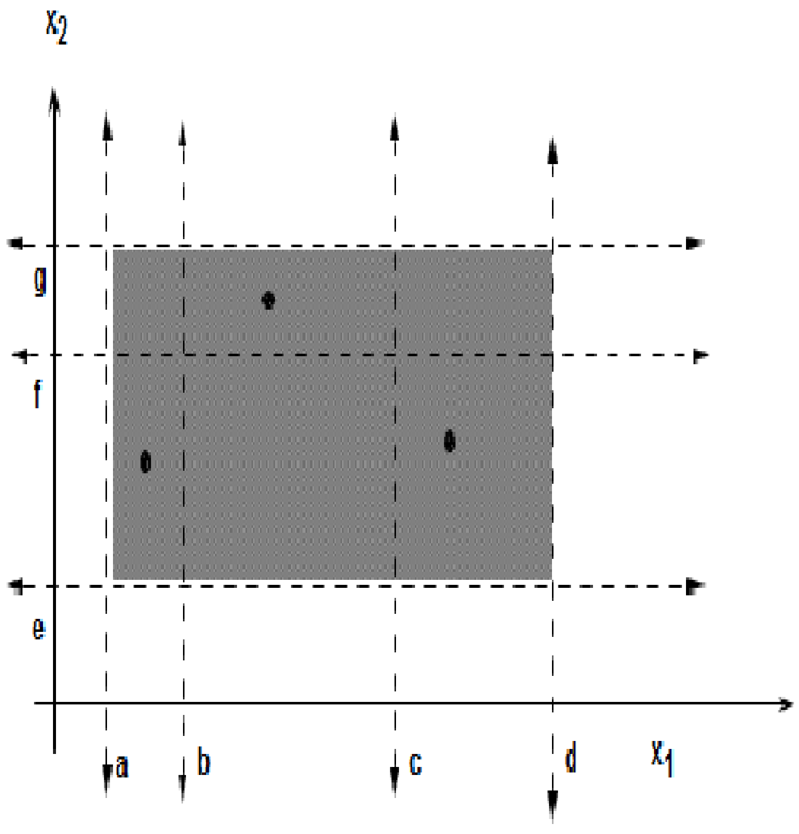
- Function F of two variable X1 & X2 ,when implemented as a program, the i/p variables X1 & X2 will have the following boundaries and intervals within the boundaries

  $a \le x1 \le d$, with intervals [a,b),[b,c),[c,d]

  $e \le x2 \le g$, with intervals [e,f),[f,g]

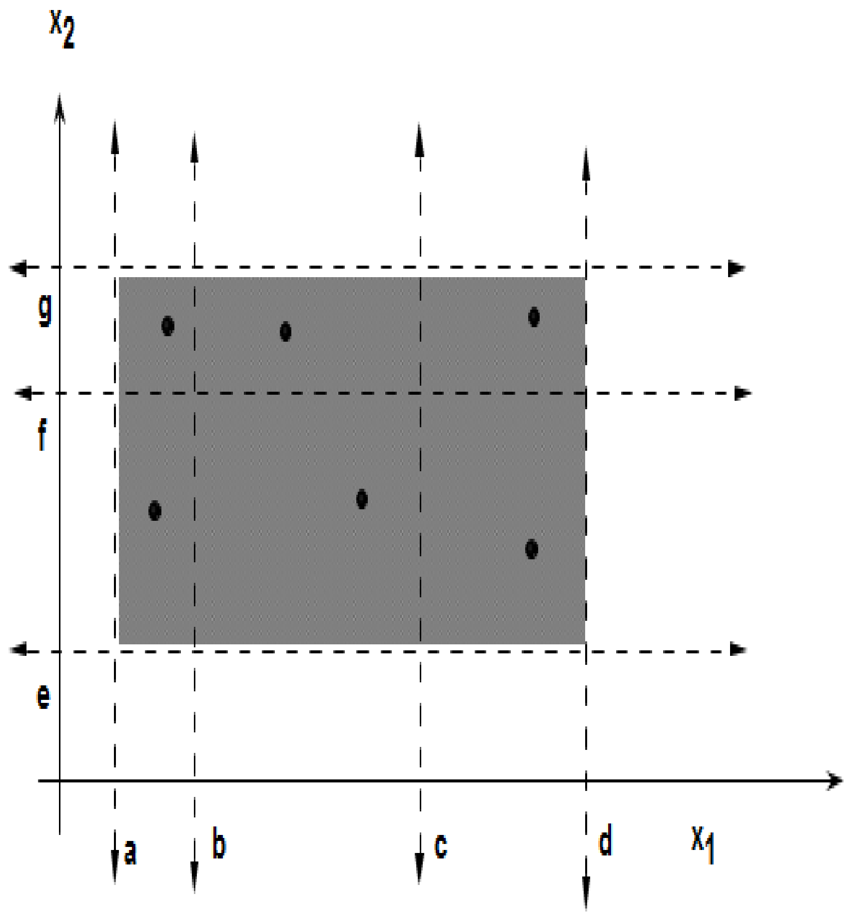
- Main purpose of Equivalence Class are:
  - To have a sense of complete testing
  - To avoid redundancy

Weak Normal Equivalence Class Testing



- ✓ Accomplished by using one variable from each equivalence class in a test cases
- ✓ These three test cases use one value from each equivalence class

Strong Normal Equivalence Class Testing

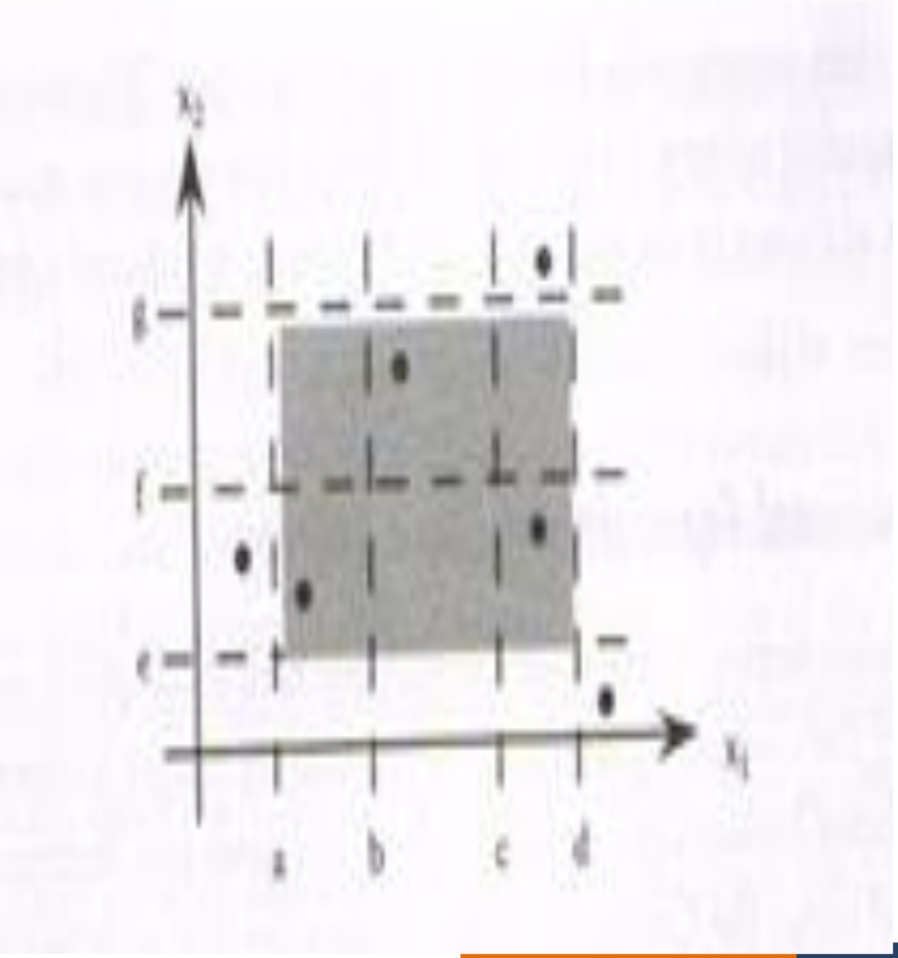


- Based on multiple fault assumption
- We need test cases from each element of the Cartesian product of the equivalence class

Strong Normal Equivalence Class Testing

- The Cartesian product guarantees that we have a notation of completeness in 2 sense:
  - ✓We cover all the equivalence classes
  - ✓We have one of each possible combination of inputs.

Weak Robust Equivalence Class Testing



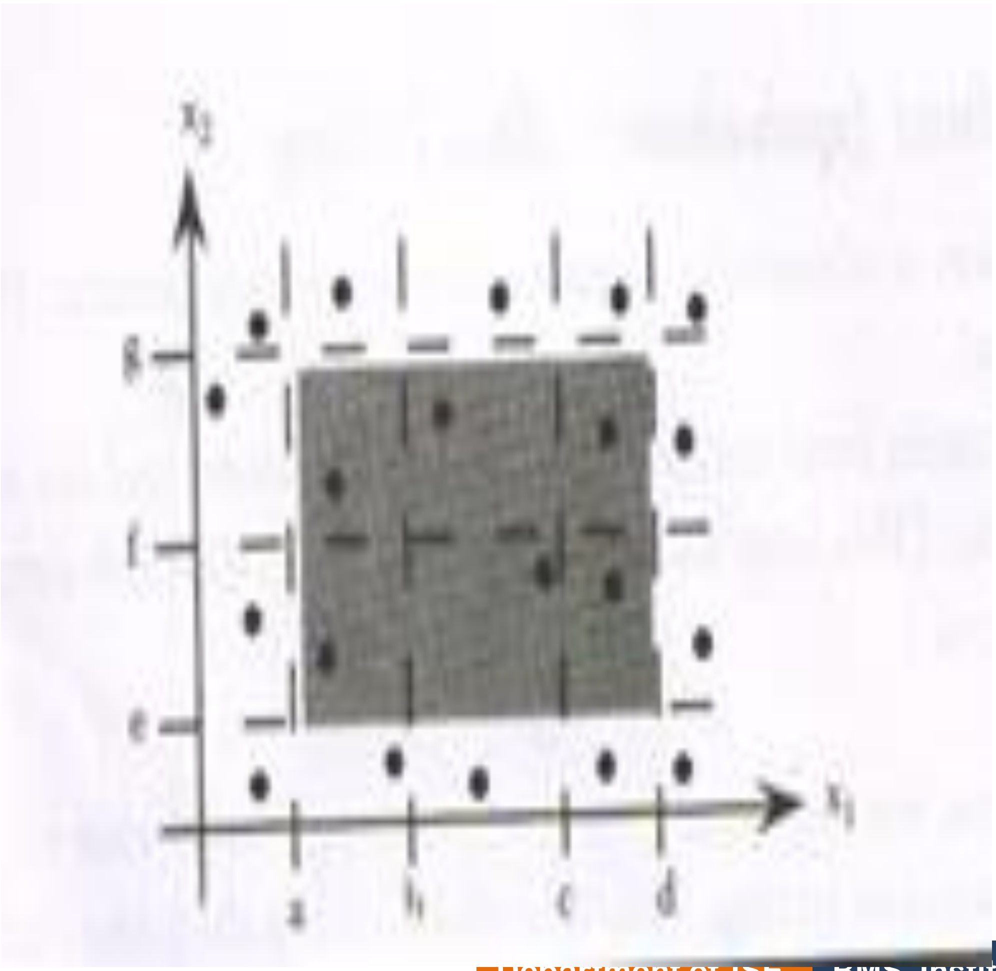
- **Robust part — Comes from consideration of invalid values**
- **Weak part — refers to the single fault assumption**

Weak Robust Equivalence Class Testing

Two problems occur

- Specification does not define what expected output for an invalid input should be. Thus testers spend a lot of time defining expected outputs for these cases.
- Strongly typed languages eliminate the need for the consideration of invalid inputs.

Strong Robust Equivalence Class Testing



The robust part -- comes from consideration
Of invalid values
Strong part -- refers to the multiple fault assumption

Weak Equivalence class Testing

Test Case	a	b	c
WE1	a1	b1	c1
WE2	a2	b2	c2
WE3	a3	b3	c3
WE4	a1	b4	c2

Strong Equivalence class Testing

Test Case	a	b	c
SE1	a1	b1	c1
SE2	a1	b1	c2
SE3	a1	b2	c1
SE4	a1	b2	c2
SE5	a1	b3	c1
SE6	a1	b3	c2
SE7	a1	b4	c1
SE8	a1	b4	c2
SE9	a2	b1	c1
SE10	a2	b1	c2
SE11	a2	b2	c2
SE12	a2	b2	c2
SE13	a2	b3	c1
SE14	a2	b3	c2
SE15	a2	b4	c1
SE16	a2	b4	c2
SE17	a3	b1	c1

Equivalence Class Test Cases for triangle problem

In the problem statement, we note that there are four possible outputs: Not a Triangle, Scalene, Isosceles, and Equilateral. We can use these to identify output (range) equivalence classes as follows.

R1 = {<a, b, c> : the triangle with sides a, b, and c is equilateral}
R2 = {<a, b, c> : the triangle with sides a, b, and c is isosceles}
R3 = {<a, b, c> : the triangle with sides a, b, and c is scalene}
R4 = {<a, b, c> : sides a, b, and c do not form a triangle}

These classes yield a simple set of test cases:

Test Case	a	b	c	Expected Output
OE1	5	5	5	Equilateral
OE2	2	2	3	Isosceles
OE3	3	4	5	Scalene
OE4	4	1	2	Not a Triangle

$$D1 = \{<a, b, c> : a = b = c\}$$
$$D2 = \{<a, b, c> : a = b, a \neq c\}$$
$$D3 = \{<a, b, c> : a = c, a \neq b\}$$
$$D4 = \{<a, b, c> : b = c, a \neq b\}$$
$$D5 = \{<a, b, c> : a \neq b, a \neq c, b \neq c\}$$

$$D6 = \{<a, b, c> : a \geq b + c\}$$
$$D7 = \{<a, b, c> : b \geq a + c\}$$
$$D8 = \{<a, b, c> : c \geq a + b\}$$

Alternately

$$D6' = \{<a, b, c> : a = b + c\}$$
$$D6'' = \{<a, b, c> : a > b + c\}$$

Equivalence Class Test Cases for NextDate function

equivalence class testing. NextDate is a function of three variables, month, day, and year, and these have ranges defined as follows:

$1 \le month \le 12$
$1 \le day \le 31$
$1812 \le year \le 2012$

Traditional Test Cases

The valid equivalence classes are

M1 = { month : $1 \le month \le 12$ }
D1 = { day : $1 \le day \le 31$ }
Y1 = { year : $1812 \le year \le 2012$ }

The invalid equivalence classes are

M2 = { month : month < 1 }
M3 = { month : month > 12 }
D2 = { day : day < 1 }
D3 = { day : day > 31 }
Y2 = { year : year < 1812 }
Y3 = { year : year > 2012 }

These classes yield the following test cases, where the valid inputs are mechanically selected from the approximate middle of the valid range:

Case ID	Month	Day	Year	Expected Output
TE1	6	15	1912	6/16/1912
TE2	-1	15	1912	invalid input
TE3	13	15	1912	invalid input
TE4	6	-1	1912	invalid input
TE5	6	32	1912	invalid input
TE6	6	15	1811	invalid input
TE7	6	15	2013	invalid input

Decision tables

- **Used to represent and analyse complex logical relationships**
- **Ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions**

Decision Table Testing (Cont....)

- **A decision table has following four portions**
  - ✓ **Stub portion – left most column**
  - ✓ **Entry portion– right**
  - ✓ **Condition portion– C's** → **Condition stub**, **condition entries**
  - ✓ **Action portion– a's** → **Action stub**, **Action entries**

Decision Table Testing (Cont….)

Table 7.1 Portions of a Decision Table

Stub	Rule 1	Rule 2	Rules 3, 4	Rule 5	Rule 6	Rules 7, 8
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	—
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

Decision Table Testing (Cont….)

- Don’t care entry has 2 major interpretation
  - ✓The condition is irrelevant
  - ✓The condition doesn’t apply

Decision Table Testing (Cont....)

- **LIMITED ENTRY DISION TABLES**
  Decision table in which all the conditions are binary
- **EXTENDED ENTRY DECISION TABLE**
  If conditions are allowed to have several values

Decision Table Testing Technique

- To identify test cases with decision tables interpret
  - ✓Conditions as inputs(refers equivalence classes of inputs)
  - ✓Actions as outputs(refers functional processing portions of the item tested)
  - ✓Rules as Test cases

Decision Table for triangle problem

Table 7.2 Decision Table for the Triangle Problem

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F
c4: b = c?	—	T	F	T	F	T	F	T
a1: Not a Triangle	X							
a2: Scalene								
a3: Isosceles					X		X	X
a4: Equilateral		X						
a5: Impossible			X	X		X		

Refined Decision Table for triangle problem

Refined Decision Table for the Triangle Problem

a < b + c?	F	T	T	T	T	T	T	T	T	T	T
b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c < a + b?	—	—	F	T	T	T	T	T	T	T	T
a = b?	—	—	—	T	T	T	T	F	F	F	F
a = c?	—	—	—	T	T	F	F	T	T	F	F
b = c?	—	—	—	T	F	T	F	T	F	T	F
Not a Triangle	X	X	X								
Scalene											X
Isosceles							X		X	X	
Equilateral				X							
Impossible					X	X		X			

Decision Table with mutually Exclusive Conditions

Table 7.4 Decision Table with Mutually Exclusive Conditions

Conditions	*R1*	*R2*	*R3*
c1: month in M1?	T	—	—
c2: month in M2?	—	T	—
c3: month in M3?	—	—	T
a1			
a2			
a3			

RULE COUNTS

- When don’t care entries really indicate that the conditions are irrelevant, rule counts are developed as follows:
  - ✓Rule in which no don’t care entries occur, count as one rule
  - ✓Each don’t care entry in a rule doubles rule count

Decision Table with rule count

Table 7.5 Decision Table for Table 7.3 with Rule Counts

c1: a < b + c?	F	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T
Rule count	32	16	8	1	1	1	1	1	1	1
a1: Not a Triangle	X	X	X							
a2: Scalene										
a3: Isosceles							X		X	X
a4: Equilateral				X						
a5: Impossible					X	X		X		

Rule counts for a decision table with mutually Exclusive conditions

Table 7.6 Rule Counts for a Decision Table with Mutually Exclusive Conditions

Conditions	R1	R2	R3
c1: month in M1	T	—	—
c2: month in M2	—	T	—
c3: month in M3	—	—	T
Rule count	4	4	4
a1			

EXPANDED VERSION

Expanded Version of Table 7.6

Conditions	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	3.4
[illegible] in M1	T	T	T	T	T	T	F	F	T	T	F	F
[illegible] in M2	T	T	F	F	T	T	T	T	T	F	T	F
[illegible] in M3	T	F	T	F	T	F	T	F	T	T	T	T
Rule count	1	1	1	1	1	1	1	1	1	1	1	1

Mutually Exclusive Conditions with Impossible Rules

	1.1	1.2	1.3	1.4	2.3	2.4	3.4	
in M1	T	T	T	T	F	F	F	F
in M2	T	T	F	F	T	T	F	F
in M3	T	F	T	F	T	F	T	F
count	1	1	1	1	1	1	1	1
impossible	X	X	X		X			X

Table 7.9 A Redundant Decision Table

Conditions	1–4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	X
a2	—	X	X	X	—	—
a3	X	—	X	X	X	X

Action entries in 9 -----identical to 1-4

An inconsistent Decision Table

Table 7.10 An Inconsistent Decision Table

Conditions	1–4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	—
a2	—	X	X	X	—	X
a3	X	—	X	X	X	—

Rule 9 -----identical to 1-4 but actions are different

An inconsistent Decision Table

Observations

- ✓Rule 1-4 and 9 are inconsistent– Action sets are different
- ✓Decision table is non deterministic-no way to decide which rule to apply

Testers should take care when don't care entries are used in decision table.

Decision Table for triangle problem

Refined Decision Table for the Triangle Problem

a < b + c?	F	T	T	T	T	T	T	T	T	T	T
b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c < a + b?	—	—	F	T	T	T	T	T	T	T	T
a = b?	—	—	—	T	T	T	T	F	F	F	F
a = c?	—	—	—	T	T	F	F	T	T	F	F
b = c?	—	—	—	T	F	T	F	T	F	T	F
Not a Triangle	X	X	X								
Scalene											X
Isosceles							X		X	X	
Equilateral				X							
Impossible					X	X		X			

Introduction:

Let's count marbles … a lot of marbles

- Suppose we have a big bowl of marbles. How can we estimate how many?
  - I don't want to count every marble individually
  - I have a bag of 100 other marbles of the same size, but a different color
  - What if I mix them?

Estimating marbles

- I mix 100 black marbles into the bowl
  - Stir well …
- I draw out 100 marbles at random
- 20 of them are black

- How many marbles were in the bowl to begin with?

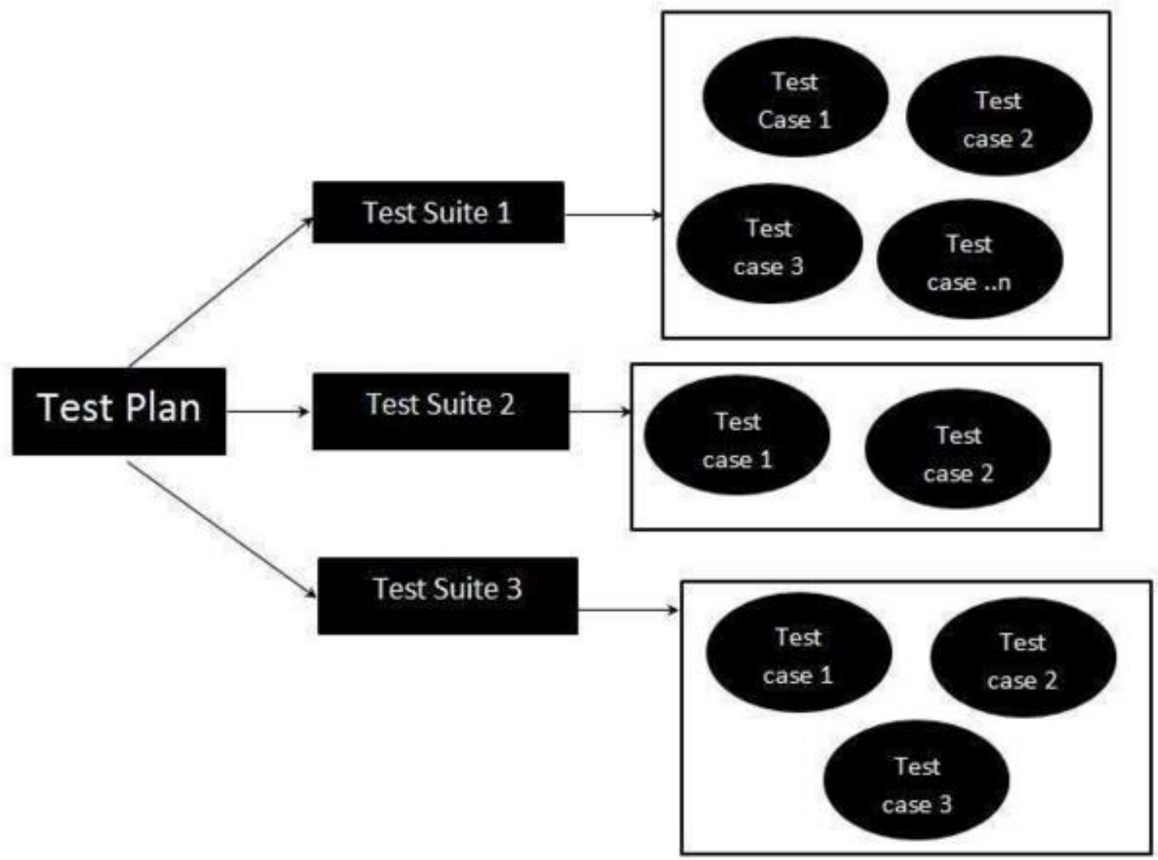
Estimating Test Suite Quality

- Now, instead of a bowl of marbles, I have a program with bugs
- I add 100 new bugs
  - Assume they are exactly like real bugs in every way
  - I make 100 copies of my program, each with one of my 100 new bugs
- I run my test suite on the programs with seeded bugs …
  - … and the tests reveal 20 of the bugs
  - (the other 80 program copies do not fail)

Test Suite

Test suite is a container that has a set of tests which helps testers in executing and reporting the test execution status. It can take any of the three states namely **Active, Inprogress and completed**.

Test Suite - Diagram:



Fault-Based Testing [TDM]

- The Basic concept of fault-based testing is to **select test cases that would distinguish the program under test** from **alternative programs that contain hypothetical faults**
- **TDM- Test Data Management**

Definition:

Fault-based testing is the process of demonstrating the absence of pre-specified faults in a module under test (MUT).

Explanation:

The definition given here has a particular focus, scope, and goal.

The focus is on **faults** rather than errors.

The scope is limited to **pre-specified faults** rather than all possible faults.

The goal is to **demonstrate the absence of faults**, not merely to look for faults (or errors).

Assumption in Fault-based Testing

- The effectiveness of fault-based testing depends on the quality of the fault model and on some basic assumptions about the relation of the seeded faults to faults that might actually be present.
- Competent programmer hypothesis.
- Coupling Effect.
- Fault based testing can guarantee fault detection only if the competent programmer hypothesis and coupling effect hypothesis hold. [TDM]

Fault-Based Testing: Terminology

Original program The program unit (e.g., C function or Java class) to be tested.

Program location A region in the source code. The precise definition is defined relative to the syntax of a particular programming language. Typical locations are statements, arithmetic and Boolean expressions, and procedure calls.

Alternate expression Source code text that can be legally substituted for the text at a program location. A substitution is legal if the resulting program is syntactically correct (i.e., it compiles without errors).

Alternate program A program obtained from the original program by substituting an alternate expression for the text at some program location.

Distinct behavior of an alternate program *R* for a test *t* The behavior of an alternate program *R* is distinct from the behavior of the original program *P* for a test *t*,if *R* and *P* produce a different result for *t*, or if the output of *R* is not defined for *t*.

Mutation Analysis

- **Mutation analysis** is the most common form of **software fault-based testing**.
- A fault model is used to produce hypothetical faulty programs by creating variants of the program under test.
- Variants are created by "seeding" faults,

  i.e making a small change to the program under test following a pattern in the fault model.
- The patterns for **changing program text** are called **mutation operators**, and each variant program is called **mutant.**

- A *mutant* is a copy of a program with a *mutation*
- A *mutation* is a syntactic change (a seeded bug)
  - Example: **change (i < 0) to (i <= 0)**
- The basic principle in **mutation testing** is that small changes are made in a module and then the original and **mutant modules are compared**.
- **Run test suite on all the mutant programs**
- A mutant is ***killed*** **if it fails on at least one test case**
- If many **mutants are killed**, **infer that the test suite is also effective at finding real bugs**

Mutant: A program with a planted fault

- Execute mutants on each member of test set Compare results.
- Mutation Adequacy Score =D/N

D=No. of dead mutants

N = No. of non equivalent mutants

```
c = a + b;        c = a – b;
    ↓                 ↓
    R1                R2
```

If (R1 = R2): mutant is alive otherwise it is killed.

```
/** Convert each line from standard input */
void transduce() {
   #define BUFLEN 1000
   char buf[BUFLEN]; /* Accumulate line into this buffer */
   int pos=0; /* Index for next character in buffer */

   char inChar; /* Next character from input */

   int atCR = 0; /* 0="within line", 1="optional DOS LF" */

   while ((inChar = getchar()) != EOF ) {
     switch (inChar) {
     case LF:
       if (atCR) { /* Optional DOS LF */
         atCR = 0;
       } else {    /* Encountered CR within line */
         emit(buf, pos);
         pos=0;
       }
       break;
     case CR:
       emit(buf, pos);
       pos=0;
       atCR = 1;
       break;
     default:
       if (pos >= BUFLEN-2) fail("Buffer overflow");
       buf[pos++] = inChar;
     }/* switch */
   }
   if (pos > 0) {
     emit(buf, pos);
   }
}
```

Open table as spreadsheet ID	Operator	Description	Constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq C$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$

Expression Modifications			
abs	absolute value insertion	replace e by abs(e)	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Mutation Operators

- Syntactic change from **legal program to legal program**
  - So: Specific to each programming language. C++ mutations don't work for Java, Java mutations don't work for Python
- Examples:
  - crp: constant for constant replacement
    - for instance: **from (x < 5) to (x < 12)**
    - select from constants found somewhere in program text
  - ror: relational operator replacement
    - for instance: **from (x <= 5) to (x < 5)**
  - vie: variable initialization elimination
    - change **int x =5; to int x;**

Fault-Based Adequacy criteria

Given a program and a test suite *T*, mutation analysis consists of the following steps:

Select mutation operators If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

Generate mutants Mutants are generated mechanically by applying mutation operators to the original program.

Distinguish mutants Execute the original program and each generated mutant with the test cases in *T* . A mutant is *killed* when it can be distinguished from the original program.

$$TS = \{1U, 1D, 2U, 2D, 2M, End, Long\}$$

Mutant can remain live for two reasons

- The mutant can be distinguished from the original program, but the test suite *T* does not contain a test case that distinguishes them (i.e., the test suite is not adequate with respect to the mutant).
- The mutant cannot be distinguished from the original program by any test case (i.e., the mutant is equivalent to the original program).

Estimating Population Sizes

- **Counting fish** Lake Winnemunchie is inhabited by two kinds of fish, a native trout and an introduced species of chub. The **Fish and Wildlife Service** wishes to estimate the populations to evaluate their efforts to eradicate the chub without harming the population of native trout.

The population of chub can be estimated statistically as follows. 1000 chub are netted, their dorsal fins are marked by attaching a tag, then they are released back into the lake. Over the next weeks, fishermen are asked to report the number of tagged and untagged chub caught. If 50 tagged chub and 300 untagged chub are caught, we can calculate

$$\frac{1000}{\text{untagged chub population}} = \frac{50}{300}$$

-

Counting residual faults

- A similar procedure can be used to estimate the number of faults in a program: Seed a given number **S of faults in the program**. Test the program with some test suite and count the number of revealed faults.
- Measure the number of seeded faults detected, *DS*, and also the number of natural faults *DN* detected. Estimate the total number of faults remaining in the program, assuming the test suite is as effective at finding natural faults as it is at finding seeded faults, using the formula

$$\frac{S}{\text{total natural faults}} = \frac{D_S}{D_N}$$

Agenda

1. Introduction
2. Fault-Based Testing [TDM]
3. Assumption in Fault-based Testing
4. Fault Based Testing – Terminologies
5. Mutation Analysis
6. Fault-Based Adequacy criteria
7. **Variations on Mutation**
8. Fault-based testing criteria
9. Test Execution
10. Scaffolding
11. Test Oracles
12. Capture & Reply
13. Conclusions

Variations on Mutation

- Weak mutation
- Statistical mutation

Weak mutation

- Problem: There are lots of mutants. Running each test case to completion on every mutant is expensive
  - Number of mutants grows with the **square of program size**
- Approach:
  - Execute meta-mutant (with many seeded faults) together with original program
  - Mark a seeded fault as "killed" as soon as a difference in intermediate state is found
    - **Without waiting for program completion**
    - **Restart with new mutant selection after each "kill"**

Statistical Mutation

- **Problem:** There are lots of mutants. Running each test case on every mutant is expensive
  - It's just too expensive to create N^2 mutants for a program of N lines (even if we don't run each test case separately to completion)
- **Approach:** Just create a random sample of mutants
  - May be just as good for *assessing* a test suite
    - **Provided we don't design test cases to kill particular mutants (which would be like selectively picking out black marbles anyway)**

In real life …

- Fault-based testing is a widely used in semiconductor manufacturing
  - With good *fault models* of typical manufacturing faults, e.g., "stuck-at-one" for a transistor
  - But fault-based testing for ***design errors* is more challenging (as in software)**
- Mutation testing is not widely used in industry
  - But plays a role in software testing research, to compare effectiveness of testing techniques
- Some use of fault models to design test cases is important and widely practiced

Mutation Analysis Procedure

1. Generate a large number of "mutant" programs by replicating the original program except for one small change (e.g., change the "+" in line 17 to a "-", change the "<" in line 132 to a "<=", etc.).
2. Compile and run each mutant program against the test set.

(cont'd)

Mutation Analysis Procedure (cont'd)

3. Compare the ratio of mutants "killed" (i.e., revealed) by the test set to the number of "survivors."

- The higher the "kill ratio" the better the test set.

Error Seeding

- A similar approach, *Error Seeding,* has been used to estimate the "number of errors" remaining in a program.
- But such metrics are inherently problematic. For example, how many "errors" are in the following Quick Sort program?

```
QSORT(X,N)
    Return(X)
END
```

Error Seeding Procedure

1. Before testing, "seed" the program with a number of "typical errors," keeping careful track of the changes made.
2. After a period of testing, compare the number of seeded and non-seeded errors detected.

(cont'd)

Error Seeding Procedure (cont'd)

3. If N is the total number of errors seeded, n is the number of seeded errors detected, and x is the number of non-seeded errors detected, the number of remaining (non-seeded) errors in the program is about

$$x(N/n - 1)$$

- What assumptions underlie this formula?
- Consider its derivation…

Derivation of Error Seeding Formula

Let X be the total number of NON-SEEDED errors in the program

Assuming seeded and non-seeded errors are equally easy/hard to detect, after some period of testing, $x:n \approx X:N$.

So,

$$X \approx xN/n$$

$$X - x \approx xN/n - x$$

$$\approx x(N/n - 1)$$ as claimed.

Fault-based testing criteria

Error Seeding

- Estimate the number of faults that remain
- Measure quality of software testing

$$r = \frac{\text{\# artificial faults detected}}{}$$

f = # of not seeded errors detected

Estimated no. of inherent faults = $(1/r)*f$

- Applicable to any testing method
- Dependent on how faults are introduced

Fault-based testing criteria

Program Mutation Testing

Mutant: A program with a planted fault

- Execute mutants on each member of test set
- Compare results
- Mutation Adequacy Score =D/N

D=No. of dead mutants

N = No. of non equivalent mutants

If (R1 = R2): mutant is alive otherwise it is killed.

Fault-based testing criteria

Variants of Program Mutation Testing

- **Weak Mutation Testing**
  - Proposed to improve efficiency
  - Mutate and test components
- **Firm Mutation Testing**
  - Select portion of program , subset of parameters and mutate them.
  - Compare original and changed versions
  - Less expensive than strong mutation testing, more efficient than weak mutation testing
  - No basis to select area of program code, parameters

Fault-based testing criteria

Criteria Inclusion Hierarchy

Strong Mutation Testing

↓

Firm Mutation Testing

↓

Weak Mutation Testing

Fault-based testing criteria

Perturbation Testing (Deviation of a system)

- Tests the robustness of a program
- Predicted fault tolerance $= \frac{\text{\# of faults detected}}{\text{total \# of executions}}$
- A perturbation function is applied to change the data state

Example:

```
int perturbation (int x)
{
        int changedX;
        changedX = x + 50;
        return changedX;
}
```

Fault-based testing criteria

Perturbation Testing

Original program

```
main()
{int x;
 x = getVal();
 if (x > 0)
    printf("X  is positive");
 else
    printf("X  is negative");
}
```

Fault injected program

```
main()
{int x;
 x = getVal();
 x = perturbation(x);
 if (x > 0)
    printf("X  is positive");
 else
    printf("X  is negative");
}
```

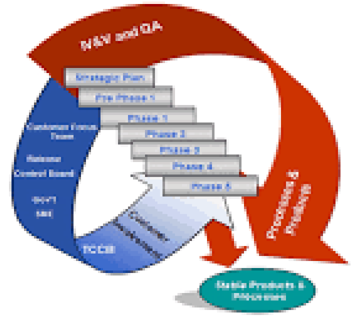
Thank you

My Details

Dr. Manjunath T. N.
Professor
Dept.of.ISE
BMSIT, Bengaluru
Email: manju.tn@bmsit.in
manju.tn@gmail.com
Mobile:+91-9900130748

Software Testing

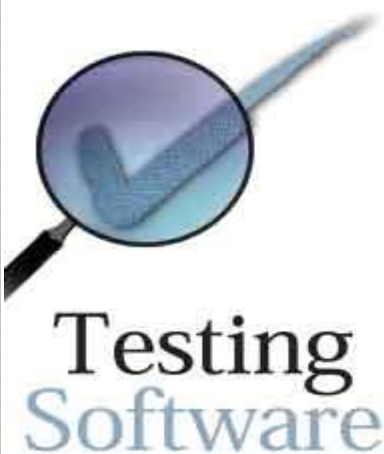
Automated Testing
Manual Testing
Software Quality Assurance



Module - 3: Structural Testing

By

Dr.Manjunath T N
Professor



Agenda

1. Overview of Structural Testing
2. Statement Testing
3. Branch Testing and Condition Testing
4. Path Testing : DD Paths
5. Test Coverage Metrics
6. Basis Path testing
7. Data Flow Testing – Define- Use Testing
8. Slice Based
9. Test Execution
10. Scaffolding
11. Test Oracles
12. Capture & Reply
13. Conclusions

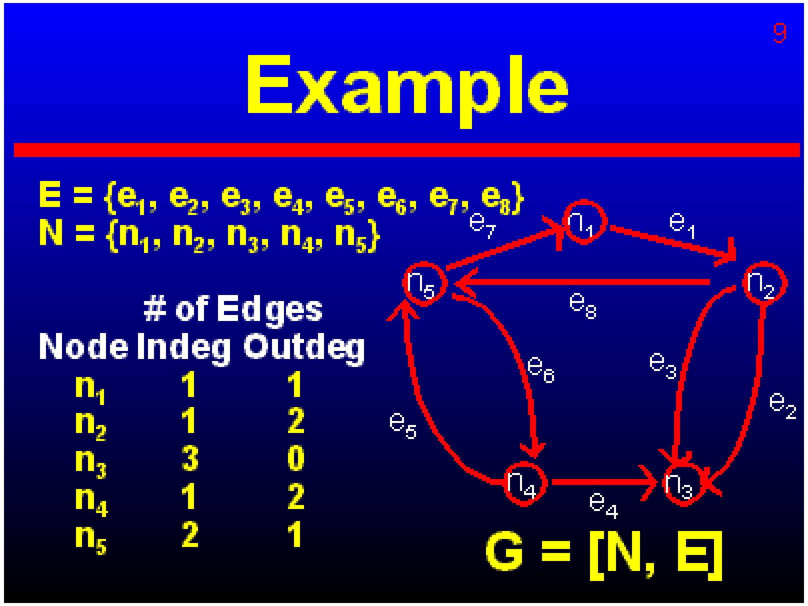
Software Testing – White Box

1.Basis Path Testing - In Lab we have Exercise 10,11 &12

2.Data Flow Testing – In Lab we have Exercise 9

Some of the Basic Definitions:

1.Graph - G(V,E)
2.Types of Graph – Directed & Undirected,
Cyclic & acyclic
3.Indegree & Out degree



Software Testing – Basis Path

Program Graph

The techniques followed for path testing start with the **program graph**

– Given a program written in an **imperative programming language**, its **program graph** is a **directed graph** in which **nodes are either entire statements or fragments of a statement**, and **edges represent flow of control**

– If i and j are nodes in the program graph, there is an edge from node i to node j if and only if the statement (fragment) corresponding to node j can be executed immediately **after the statement (fragment) corresponding to node i.**

Software Testing – Basis Path

Statements fragment examples:

Begin / End

- convenient to have those as fragments
- Some argue that they are not always better to be fragments (e.g. then begin), there is no problem in this case when the graph is composed

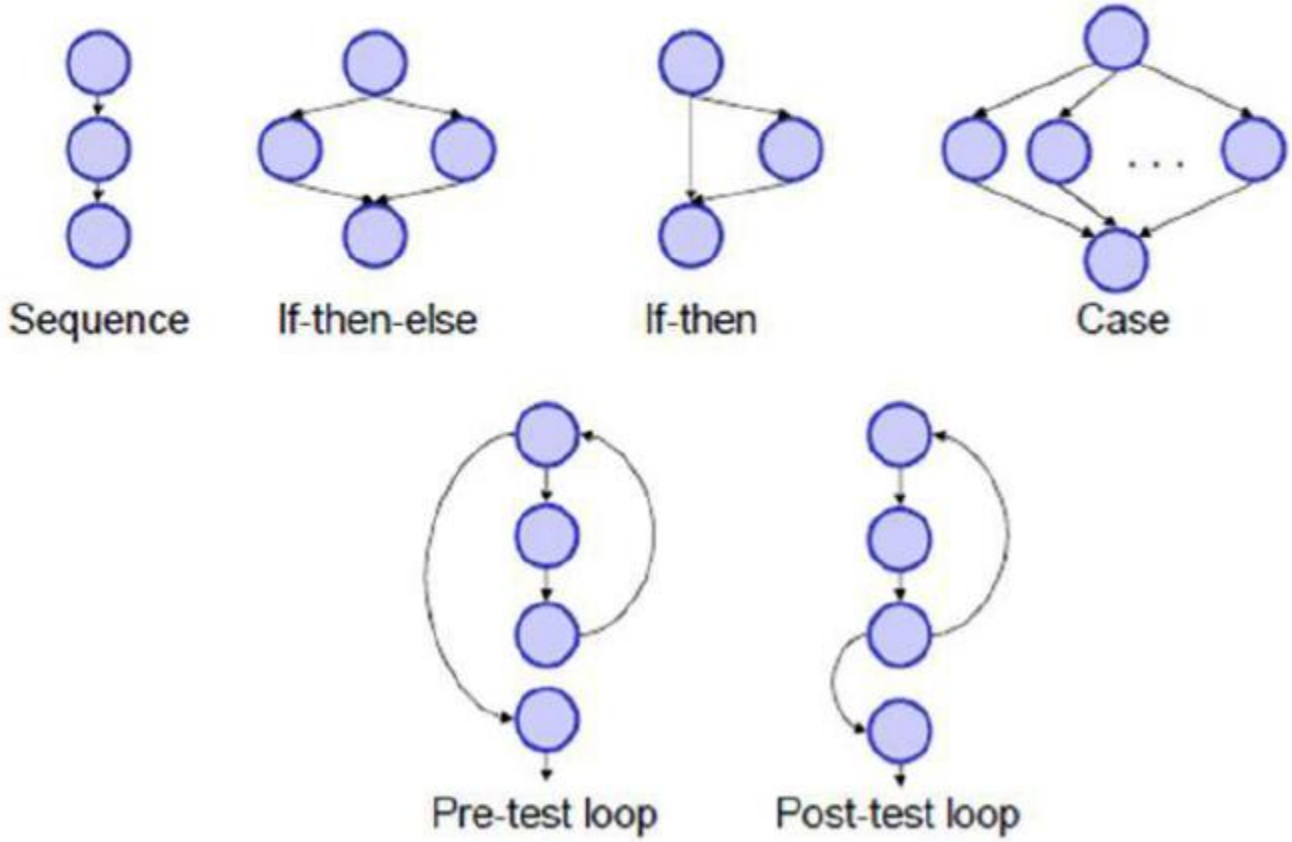
The importance of a **program graph** is that **program executions correspond to paths from the source to the sink nodes**.

Test cases force the execution of some program path

Software Testing – Basis Path

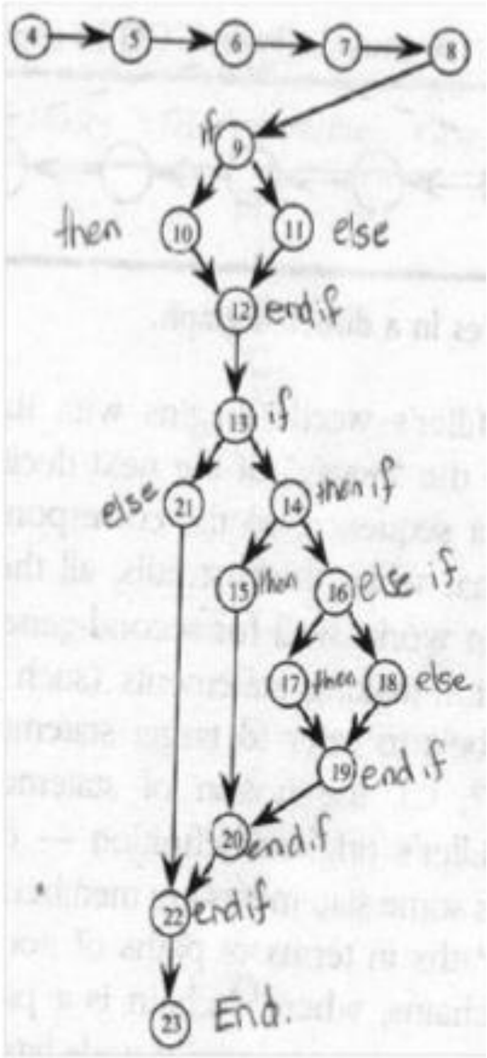
Flow Graph Notation

Notation for representing control flow



Constructing a program graph from a given program based on above notations is easy.

```
1.  Program triangle
2.  Dim a,b,c as integer
3.  Dim isatriangle is boolean
4.  Output("enter 3 sides of a triangle")
5.  Input(a,b,c)
6.  Output("side a is",a)
7.  Output("side b is",b)
8.  Output("side c is",c)
9.  If(a<b+c) and (b<a+c) and (c<a+b)
10. Then  isatriangle= true
11. Else isatriangle=false
12. Endif
13. If istriangle
14. Then if (a=b) and (b=c)
15. Then  output("equilateral")
16. Else if(a not= b) and (a not=c) and (b not=c)
17. Then output("scalene")
18. Else output("isosceles")
19. Endif
20. Endif
21. Else output("not a triangle")
22. Endif
23. End triangle
```



Cont..

Observations

- Nodes 4-8 are a sequence, there is no branching
- Nodes 9-12 are an IF-THEN-ELSE construct
- Nodes 13-22 are nested IF-THEN-ELSE constructs
- Nodes 4 and 23 are the program source and sink nodes
  - Single entry, single exit
- There are no loops, so this is a directed acyclic graph

Decision – To – Decision Paths (DD-Paths)

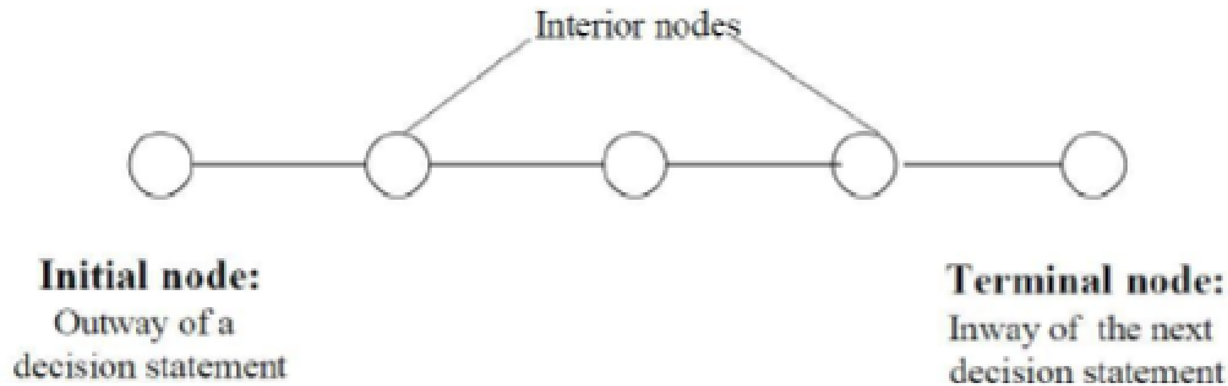
The best known form of structural testing is based on decision-to-decision path.
A DD-paths is a sequence of statements that begins with the "outway" of a decision statement and ends with the "inway" of the next decision statement.
– There are no internal branches in such a sequence
– Like a row of dominos

We will define paths in terms of nodes in a directed graph
Paths = chains
•**Chain:**
– a path in which the initial & terminal nodes are distinct
– every interior node has indegree = 1 and outdegree = 1
– A chain can consist of only one node & no edges
– Length of chain is the number of edges

•Every statement in a program is a member of **one and only one DD-Path**

•The objective is to scan the **program to break it into a number of unique DD-paths, and use each of those paths as a node to build a DD-Path Graph**.
•DD-Paths enable very precise descriptions of **test coverage**

Cont…



A chain of nodes in a directed graph of length =4

The length of the chain= the number of edges

DD-Path Definition

Definition

A *DD-Path is a sequence of nodes in a program graph such that*

Case 1: it consists of a single node with in-degree = 0

– This is the ***source node (the initial DD-path)***

Case 2: it consists of a single node with out-degree = 0

– This is the ***sink node (the final DD-path)***

Case 3: it consists of a single node with in-degree >=2 OR out-degree >=2

– Assures that no node is contained in more than 1 DD-path

Case 4: it consists of a single node with in-degree = 1 AND outdegree = 1

– Needed for short branches

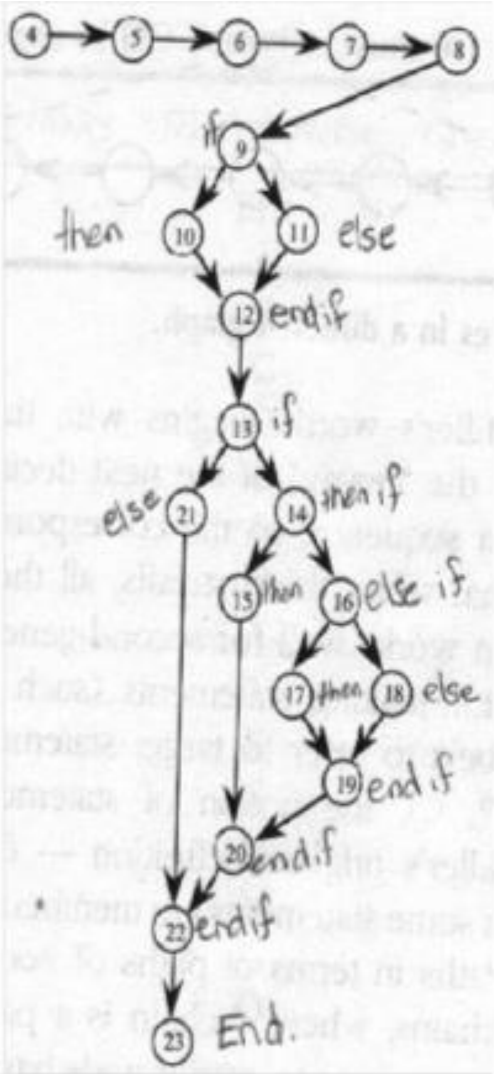
Case 5: it is a maximal chain of length >=1

– Normal case: single entry, single exit sequence of nodes

– Each node is 2-connected to every other node

» i.e. there is a path from node ni to nj (& not the reverse)

Program Graph & DD-Path



Refer the program graph of triangle program

Program Graph Nodes	DD-Path Name	Case of Definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

DD-Path Graph

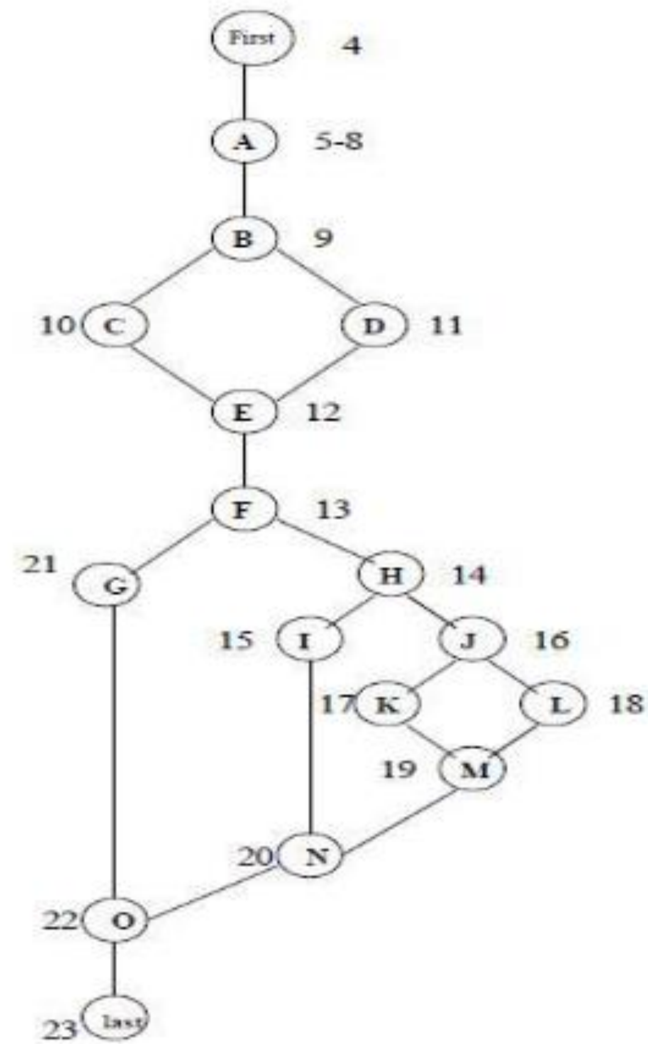
DD-Path Graph:

Definition: Given a program written in an imperative language, its DDPath graph i directed graph in which nodes are DDPaths of its program graph, and edges represent cc flow between successor DD-Paths.

The DD-Path graph is a form of condensation graph, in this condensation:

– 2-connected components are collapsed into individual nodes that correspond to case 5 Paths

– Single node DD-paths (cases 1-4) are required to assure that every statement is in ex one DD-Path



DD-Path graph for Triangle program

Cont…

Steps to Build a DD-Path Graph

1. Number the program statements and/or fragments
2. Draw a program graph
3. Divide program graph into DD-paths
   - Identify which program graph nodes form each DDpath (according to the 5 cases in the definition of DD-paths)
   - Name each DD-path (A,B, etc.)
4. Build the DD-Paths Graph

Test Coverage Metrics

Test Coverage Metrics

The raison d'etre of DD-Paths is that they enable very precise description of test coverage.

One of the fundamental limitations of functional testing is that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercise a program.

Venn diagram showing relationship among specified, programmed and tested behaviours. Test coverage metrics are a device to measure the extent to which a set of test cases covers a program.

"Test Coverage metrics are a device to measure the extent to which a set of test cases cavers a program"

Several widely accepted test coverage metrics are used; most of those in a below Table

Table: Structural test coverage metrics

Metric	Description of coverage
C_0	Every statement
C_1	Every DD-Path (predicate outcome)
C_{1p}	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
C_d	C_1 coverage + every dependent pair of DD-Paths
C_{MCC}	Multiple condition coverage
C_jk	Every program path that contains up to k repetitions of a loop (usually k=2)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

Predicate = statement fragment

Most quality organizations now expect the C1 metric as the minimum acceptable level of test coverage

There are always fault types that can be revealed at one level and can escape detection by inferior levels of testing

Metric Based Testing

Metric Based Testing

Metric based testing are techniques that exercise source code in terms of the structural test coverage metrics. These coverage metrics require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once

1. Statement and predicate testing:

- We allow statement fragments to be individual nodes
  - Hence, levels C0 & C1 collapse into 1 level
  - Nodes 8,9,10 are a complete IF-THEN-ELSE statement, if we follow C0, we could only execute one of the decision alternatives and satisfy the statement coverage criteria
  - When the statement is divided into fragments, we could do predicate outcome coverage

2. DD-Path testing

- When the C1 metric is exercised, we traverse every *edge* of the DD-Path, and thus every fragment, as opposed to every *node*.
  - For IF-THEN-ELSE statements, the true and the false branches are covered (C_{1p})
  - For CASE statements, every clause is covered
- Longer DD-Paths generally represent complex computations
  - We could consider each one of those an individual function
  - It may be appropriate to apply a number of functional tests like boundary and special value tests.

Cont…

3. Dependent Pairs of DD-Paths

- The most common dependency among pairs of DD-Paths is the define/reference relationship (define/use)
  - Where a variable is defined and could receive a value in one DD-Path and is referenced in another DD-Path.

4. Multiple Condition Coverage

- **Node B corresponds to statement 9 in the program graph, line 9:**
  IF (a < b+c) AND (b < a+c) AND (c < a+b)

  – Node H corresponds to statement 14 in the program graph: IF (a=b) AND (b=c)

  – Rather than simply traversing such predicates to their TRUE and FALSE outcomes, we could investigate the different ways that each outcome can occur

Cont...

– **There is a tradeoff between statement complexity versus path complexity**

– **Multiple Condition Coverage assures that this complexity isn't swept under the DD-Path coverage rug!**

5. Loop coverage

Loops are a highly prone portion of source code

– Types of loops

1. Concatenated loops: a sequence of disjoint loops

2. Nested loops: one is contained inside the other3. Knotted loops: are Horrible loops!, when it is possible to

3. Knotted loops: Branch into or out from the middle of a loop, and these branches are internal to other loops

Loop Testing

· every loop involves a decision, and we need to test both outcomes of the decision (traverse loop or exit)

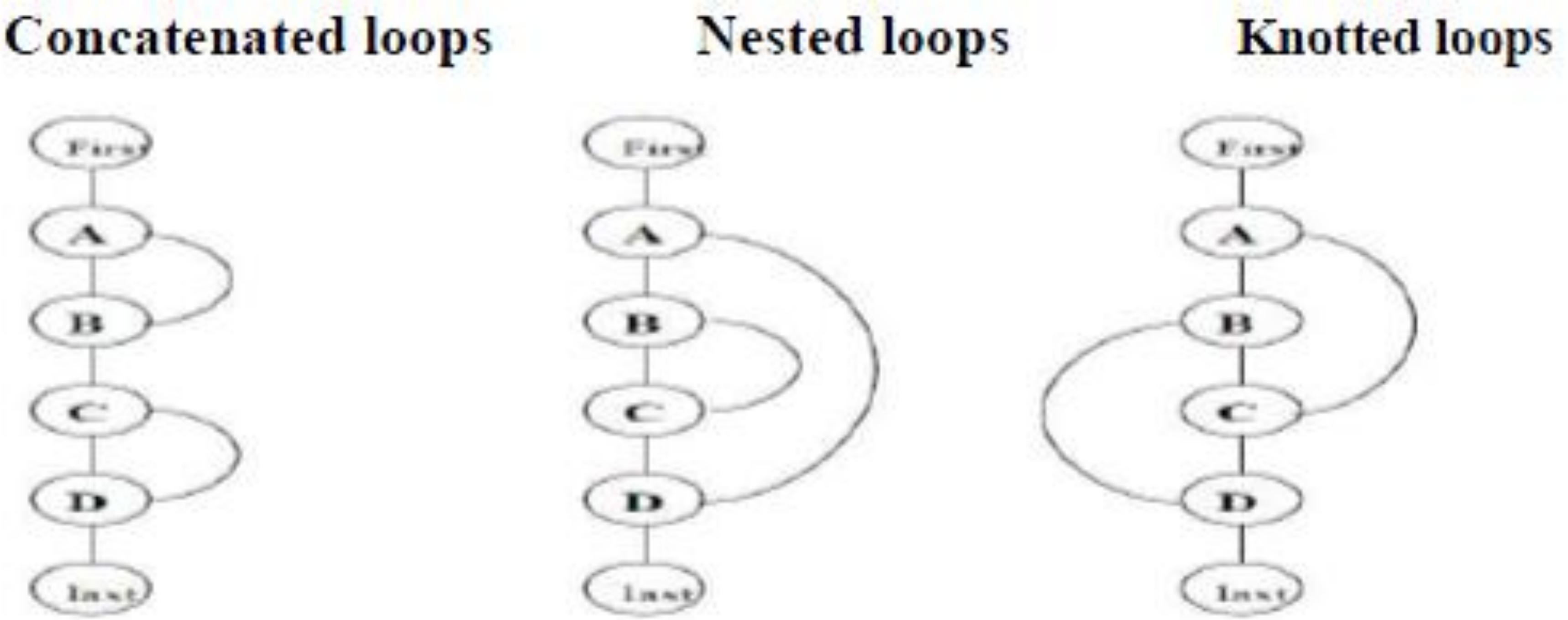
– We could do boundary value analysis on the index of the loops, or robustness testing

– F the body of a simple loops is a DD-Path that performs a complex calculation, functional testing could also be used

– Once a loop has been tested, it should be condensed into a single node

– If loops are nested, this process is repeated starting with the innermost loop and working outward.

Cont…


Concatenated loops
Nested loops
Knotted loops
First
A
B
C
D
last
First
A
B
C
D
last
First
A
B
C
D
last

Basis Path Testing

Basic Path Testing

- Mathematically usually define a **basis** in terms of a **structure** called a **vector space**, which is a set of elements (called vectors) as well as operations that correspond to multiplication and addition defined for the vectors.
- The basis of a vector space is a set of vectors that are independent of each other and span the entire vector space.

McCabe's Basis Path Testing

- A testing mechanism proposed by McCabe
- Aim is to derive a **logical complexity measure** of a **procedural design** and use this as a guide for defining a basic set of **execution paths**.
- An **execution path** is a set of nodes and directed edges in a flow graph that connects (in a directed fashion) the start node to a terminal node.

Two **execution paths** are said to be **independent** if they do **not include the same set of nodes and edges**

Cont..

Independent paths

Lesson: Paths must be feasible

Generating independent paths

- Generate one feasible path (a "baseline" path)
- Generate further paths by "flipping" each decision point in turn
  - Decision point is a node with outdegree ≥ 2
  - "Flipping" is taking a different edge than those taken previously
  - A "technically" feasible path may not be feasible "logically" (according to the logic of the program)

Basis Path Testing

Basis path testing is a hybrid between path testing and branch testing.

- Path Testing: Testing designed to execute all or selected paths through a computer system.

- Branch Testing: Testing designed to execute each outcome of each decision point in a computer program

Cont...

Figure a is taken from [McCabe 82]; it is a directed graph which we might take to be the program graph of some program, the original notation for nodes and edges is repeated here. (Notice that this is not a graph derived from a structured program: nodes B and C are a loop with two exits, and the edge from B to E is a branch into the IF-THEN statement in nodes D, E, and F. The program does have a single entry (A) and a single exit (G).) McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number of a strongly connected graph is the number of linearly independent circuits in the graph.

We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. (Notice that, if the single entry, single exit precept is violated, we greatly increase the cyclomatic number, because we need to add edges from each sink node to each source node.)

Figure 2 shows the result of doing this; it also contains edge labels that are used in the discussion that follows.

There is some confusion in the literature about the correct formula for cyclomatic complexity. Some sources give the formula as $V(G) = e - n + p$, while others use the formula $V(G) = e - n + 2p$

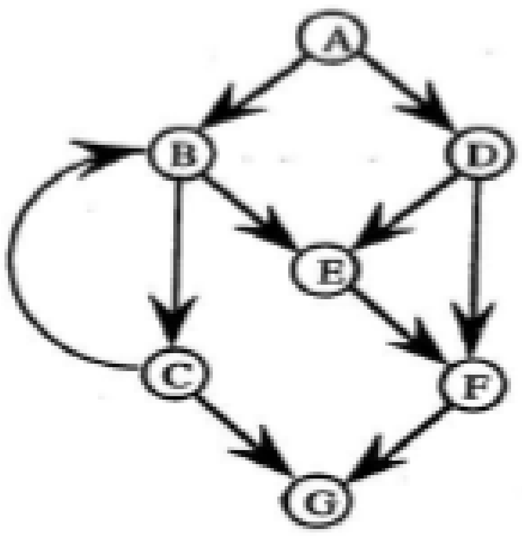
Cont…

Where E = number of edges in G

N = number of nodes in G and

P = number of connected regions

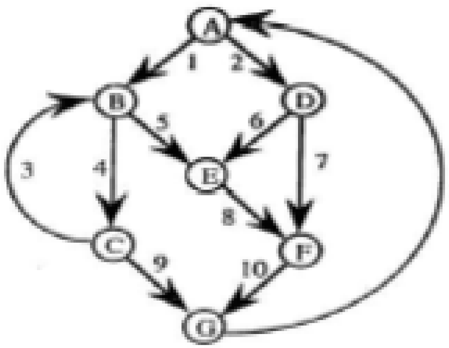
The number of linearly independent paths from the source node to the sink node in Figure a:



V(G)= e-n+2(p) = 10-7+2(1)=5

Cont…

Figure b : For Strongly connected graph We use V(G) = e - n + p



V(G)= e-n+p = 11-7+1=5

The cyclomatic complexity of the strongly connected graph in Figure b is 5, thus there are five linearly independent circuits. If we now delete the added edge form node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths. Here we identify paths as sequences of nodes:

p1: A, B, C, G

p2: A, B, C, B, C, G

p3: A, B, E, F, G

p4: A, D, E, F, G

p5: A, D, F, G

Cont…

Table path \ edges traversed

p1: A, B, C, G
p2: A, B, C, B, C, G
p3: A, B, E, F, G
p4: A, D, E, F, G
p5: A, D, F, G

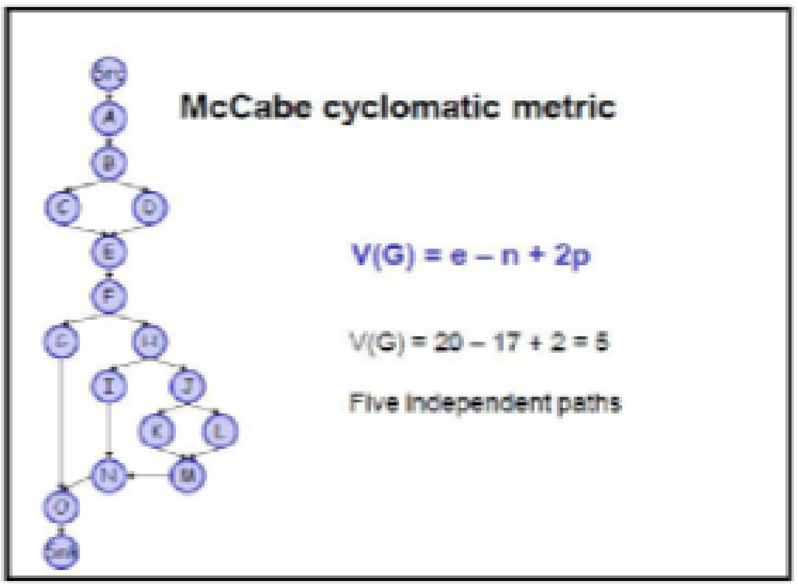
path \ edges traversed	1	2	3	4	5	6	7	8	9	10
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	1	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Cont…

Observation on McCabe's Basis Path Method

If you had trouble following some of the discussion on basis paths and sums and products of these, you may have felt a haunting skepticism, something along the lines of "Here's another academic oversimplification of a real-world problem". Rightly so, because there are two major soft spots in the McCabe view: one is that testing the set of basis paths is sufficient (it's not), and the other has to do with the yoga-like contortions we went through to make program paths look like a vector space. McCabe's example that the path A, B, C, B, C, B, C, G is the linear combination 2p2 - p1 is very unsatisfactory. What does the 2p2 part mean? Execute path p2 twice? (Yes, according to the math.) Even worse, what does the - p1 part mean? Execute path p1 backwards? Undo the most recent execution of p1? Don't do p1 next time? Mathematical sophistries like this are a real turn-off to practitioners looking for solutions to their very real problems.

To get a better understanding of these problems, we'll go back to the triangle program example.



Cont…

Independent paths for triangleDD-PATH by Base line path method

Note:- Base line path means it should contain more decision nodes (if node out degree >=2 is called decision node)

Original	P1-First- A-B-C-E-F-H-J-K-M-N-O-LAST	Scalene
Flip P1 at B	P2- First-A-B-D-E-F-H-J-K-M-N-O-LAST	Infeasible
Flip P1 at F	P3- First-A-B-C-E-F-G -O-LAST	Infeasible
Flip P1 at H	P4- First-A-B-C-E-F-H-I-N-O-LAST	Equilateral
Flip P1 at J	P5- First-A-B-D-E-F-H-J-L-M-N-O-LAST	Isosceles

If node C is traversed, then we must traverse node H.
If node D is traversed, then we must traverse node G.

Taken together, these rules, in conjunction with McCabe's baseline method, will yield the following feasible basis path set. Notice that logical dependencies reduce the size of a basis set when basis paths must be feasible.

p1: A-B-C-E-F-H-J-K-M-N-O-Last	Scalene
p6: A-B-D-E-F-G-O-Last	Not a triangle
p4: A-B-C-E-F-H-I-N-O-Last	Equilateral
p5: A-B-C-E-F-H-J-L-M-N-O-Last	Isosceles

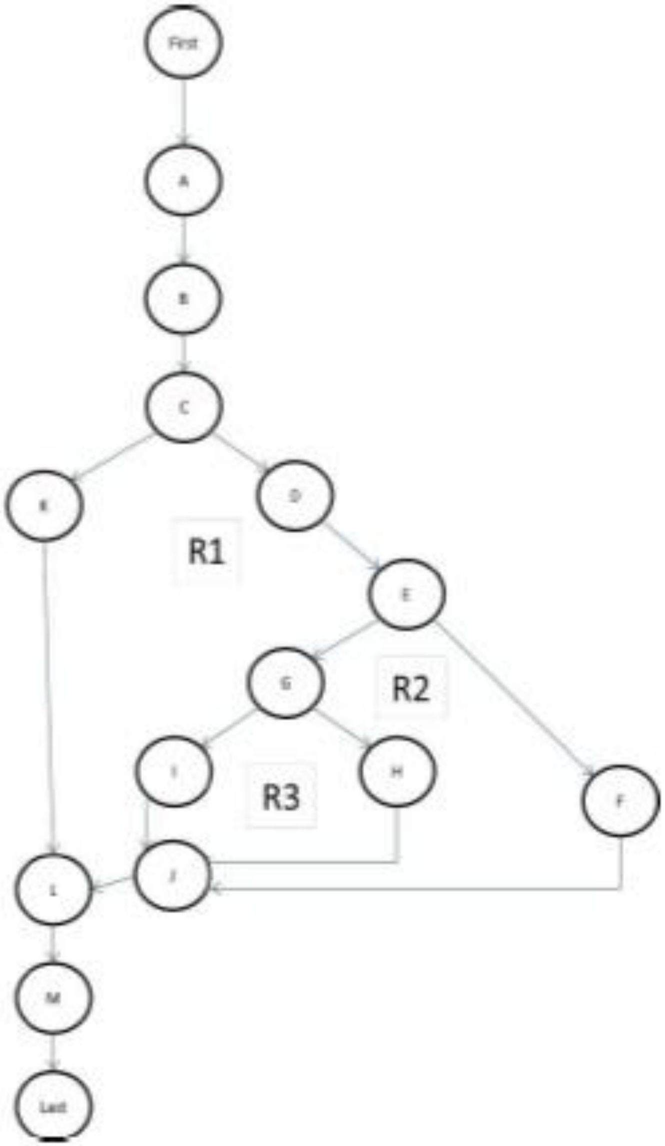
Cont...

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
#include<process.h>
int main()
{
	int a, b, c;
	clrscr();
	printf("Enter three sides of the triangle");
	scanf("%d%d%d", &a, &b, &c);
	if((a<b+c)&&(b<a+c)&&(c<a+b))
	{
		if((a==b)&&(b==c))
		{
			printf("Equilateral triangle");
		}
		else if((a!=b)&&(a!=c)&&(b!=c))
		{
			printf("Scalene triangle");
		}
		else
			printf("Isosceles triangle");
	}
	else
	{
		printf("triangle cannot be formed");
	}
getch();
return 0;
}
```

Program Graph for Triangle Program

Cont…

DD-Path Names	Program Graph Nodes
First	5
A	6,7,8,9
B	10
C	11
D	12
E	13
F	14,15,16
G	17
H	18,19,20
I	21,22
J	23
K	24,25,26,27
L	28
M	29
Last	30



Cont...

Calculation of Cyclomatic Complexity V(G) by three methods

Method 1: The cyclomatic complexity of a connected graph is provided by the formula V(G) = e – n + 2. The number of edges is represented by e, the number of nodes by n. If we apply this formula to the graph given below, the number of linearly independent circuits is:

Number of edges = 16

Number of nodes = 14

16 – 14 + 2 = 4

Method 2: V(G) = P + 1 (Where P – No. of predicate nodes with out degree = 2)

V(G) = 3 + 1 = 4. (C, E, G are the predicate nodes)

Method 3: V(G) = Number of enclosed regions + 1

V(G) = 3 + 1 = 4 (R1, R2, R3)

According to cyclomatic complexity 4 feasible basis path exists:

```
P1= First,A,B,C,D,E,F,J,L,M,Last          Equilateral
P2= First,A,B,C,D,E,G,H,J,L,M,Last        Scalene
P3= First,A,B,C,D,E,G,I,J,L,M,Last        Isosceles
P4= First,A,B,C,K,L,M,Last                Not a Triangle
```

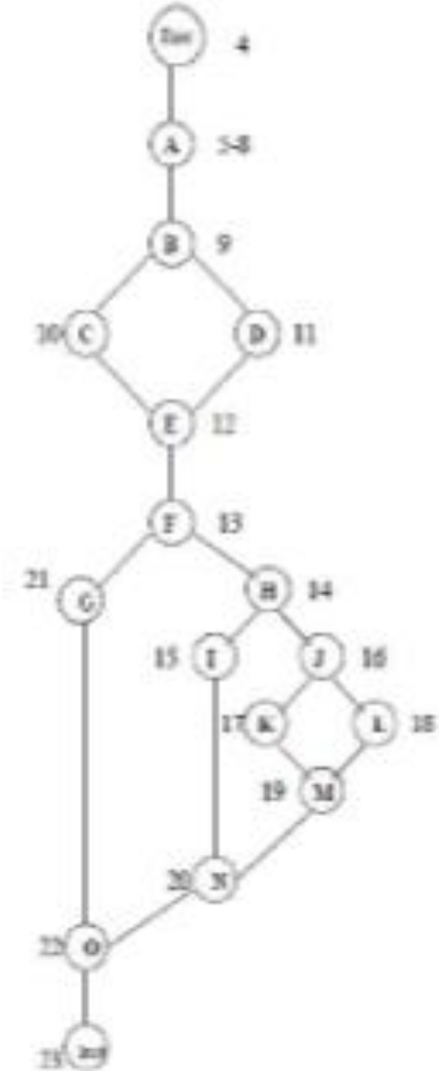
Cont…

The last step is to devise test cases for the basis paths. Test cases

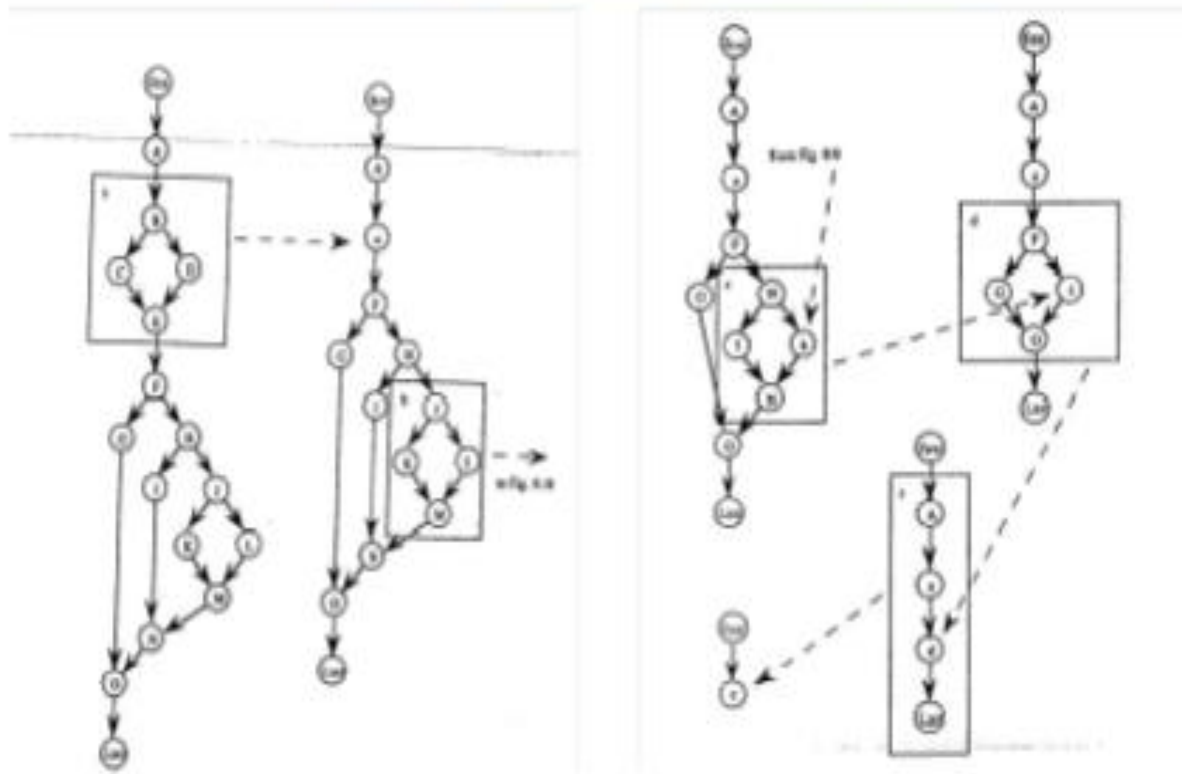
TC ID	Test Case Description	Input			Expected Output	Actual Output	Status
		A	B	C			
1	Testing for requirement 1 Path P1-To check for Equilateral Triangle	6	6	6	Equilateral Triangle		
2	Testing for requirement 2 Path P2 -To check Scalene Triangle	6	6	5	Scalene Triangle		
3	Testing for requirement 2 Path P3- To check Isosceles Triangle	5	6	7	Isosceles Triangle		
4	Testing for requirement 2 Path P4- To Check Not a triangle	1	1	2	Not a triangle		

Cont…

Example of Cyclomatic Complexity



Condensing with respect to structured programming constructs



Data Flow Testing

DATA FLOW TESTING

Data flow testing focuses on the points at which variables receive values and the points at which these values are used (or referenced). It detects improper use of data values (data flow anomalies) due to coding errors.

Rapps and Weyukers Motivation*: " it is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of Using the value produced by each and every computation.

Early data flow analyses often cantered on a set of faults that are known as define/reference anomalies.

- A variable that is defined but never used (referenced)
- A variable that is used but never defined
- A variable that is defined twice before it is used

Data flow testing
1. Define / Use Testing
2. Slice-Based Testing

DEFINE/USE TESTING

DEFINE/USE TESTING

The following refers to program P that has program graph G (P) and the set of program variables V. In a program graph statement fragments are nodes and edges represents node sequence .G (P) has single entry and single exit node. We also disallow edges from node to itself. The set of all paths in P is PATHS (P).

Definition:

– Node n ∈G(P) is a ***defining node*** of the variable v ∈ V, written as DEF(v,n), iff the value of the variable v is defined at the statement fragment corresponding to node n.

- For example: input , assignment, loop control statements (for int i=0;i<10;i++) and procedure calls are defining nodes
- When the code corresponding to such statements executes, contents of the memory location associated with v is changed

– Node n ∈G(P) is a ***usage node*** of the variable v ∈ V, written as USE(v,n), iff the value of the variable v is used at the statement fragment corresponding to node n.

- For example: output, assignment (i:=i+1), condition, loop control Statements and procedure calls are usage nodes
- When the code corresponding to such statements executes, contents of the memory location associated with v is not changed

– A usage node USE(v,n) is a ***predicate use*** (denoted as P-use), iff the statement n is a predicate statement; otherwise USE(v,n) is a ***computation use***, (denoted C-use)

- Nodes corresponding to predicate uses always have an outdegree ≥ 2
- Nodes corresponding to computation uses always have outdegree ≤ 1

– A ***definition-use (sub) path*** with respect to a variable v (denoted du-path) is a (sub) path in PATHS(P) such that for some v ∈ V, there are define and usage nodes DEF(v,m) & USE(v,n) such that m & n are the initial and final nodes of the (sub) path.

Du-path - Definition

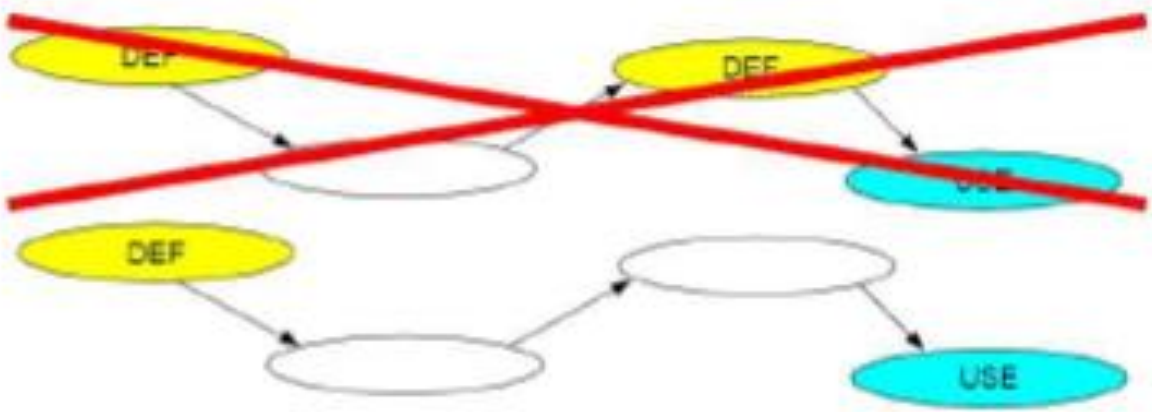
Definition of du-path:-

- **Definition-use (du) path (wrt. variable v)**
- A path in PATHS(P) such that
- for some v in V
- There exist DEF(v, m), USE(v, n) nodes s.t.
- m and n are **initial and final nodes** of the path respectively.

Dc- path - Definition

Definition of dc path:-

- **Definition-clear (dc) path (wrt. variable v)**
- A ***du-path*** in PATHS(P) where
- the initial node of the path is the **only defining node** of v (in the path).



Definition of P-use , C-use

A usage node USE(v,n) is predicate use denoted as P-case , if statement n is predicate statement (example if a<2)

If statement n is computation statement is denoted as C-case (example C=C+2)

Commission Problem

Example :-(Commission problem)

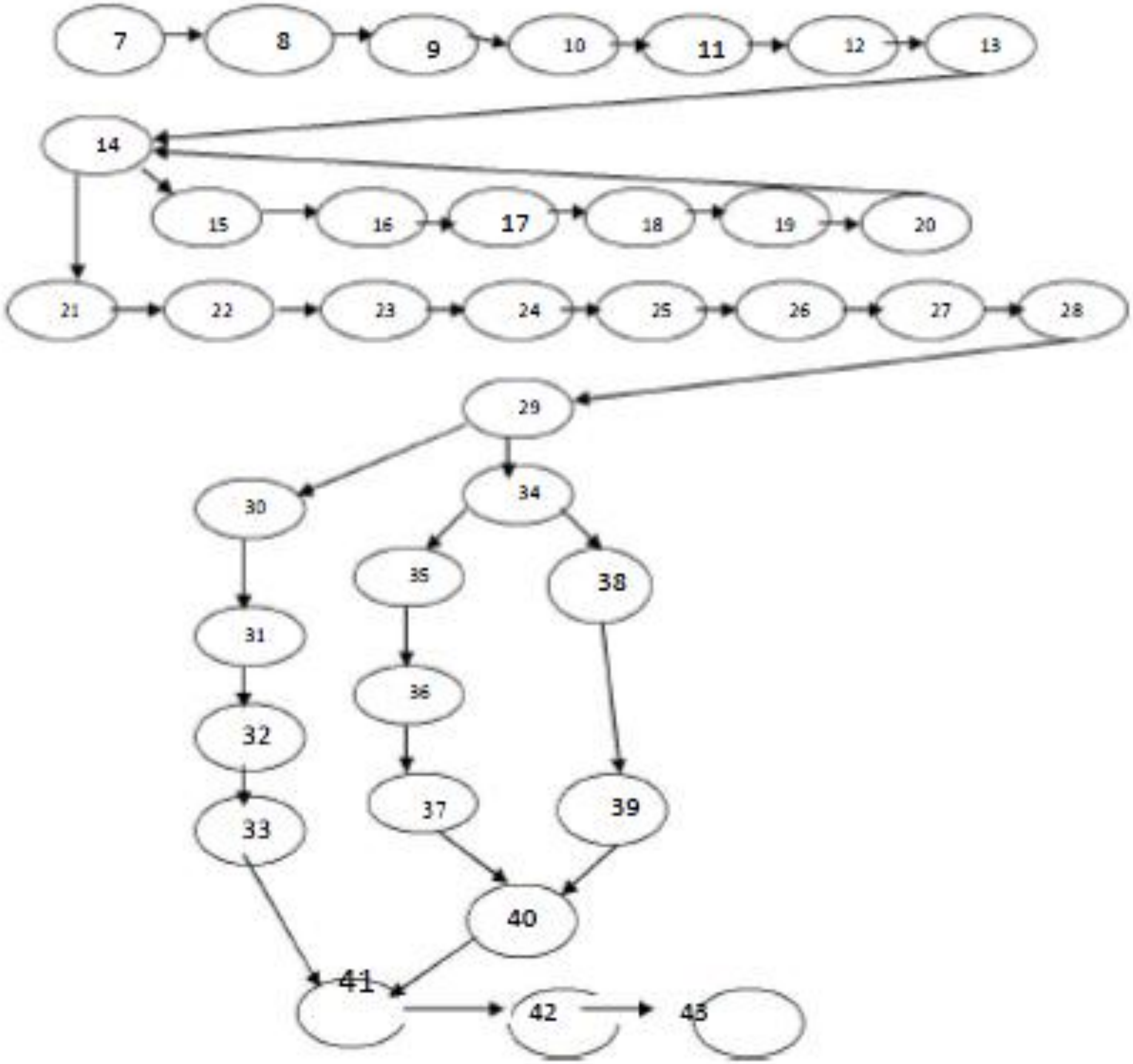
1. program commission(INPUT, OUTPUT)
2. Dim lock , stock , barrels as Integer
3. Dim lockprice ,stockprice , barrelprice As Real
4. Dim totalLocks ,totalStocks , totalBarrels As Integer
5. Dim lockSales, stocksales ,barrelsSales As Real
6. Dim sales , commission As Real
7. lockprice=45.0
8 stockprice= 30.0
9. barrelprice = 25.0
10. totallocks=0
11. totalstocks=0
12. totalbarrels=0
13 .input(locks)
14.Whle not (locks= -1)
15.Input (stock,barrel)
16.Totallocks =total locks +locks
17. Totalstocks =total stocks +stocks
18. Totalbarrels = totalbarrels +barrels
19. input (locks)
20.End While
21. output ("Locks sold", total locks)
22. output ("Stocks sold", total stocks)
23. output ("Barrels sold", totalbarrels)
24.locksales= lockprice *totallocks
25.stocksales= stockprice *totalstocks
26.barrelsales= barrelprice *totalbarrels

Cont...

```
sales= locksales + stocksales+barrelsales
output( "Totalsales", sales)
if (sales > 1800.0)
then
commission = 0.10 * 1000
commission =commission +0.15 *800.0
commission = commission +0.20 *( sales >1000)
Else if (sales >1000)
Then
commission = 0.10 * 1000.0
commission = commission +0.15 *(sales-1000.0)
Else
commission =0.10 * sales
End If
End If
output ("commission is $", commission)
End commission.
```

Cont...

Program graph of the the commission problem

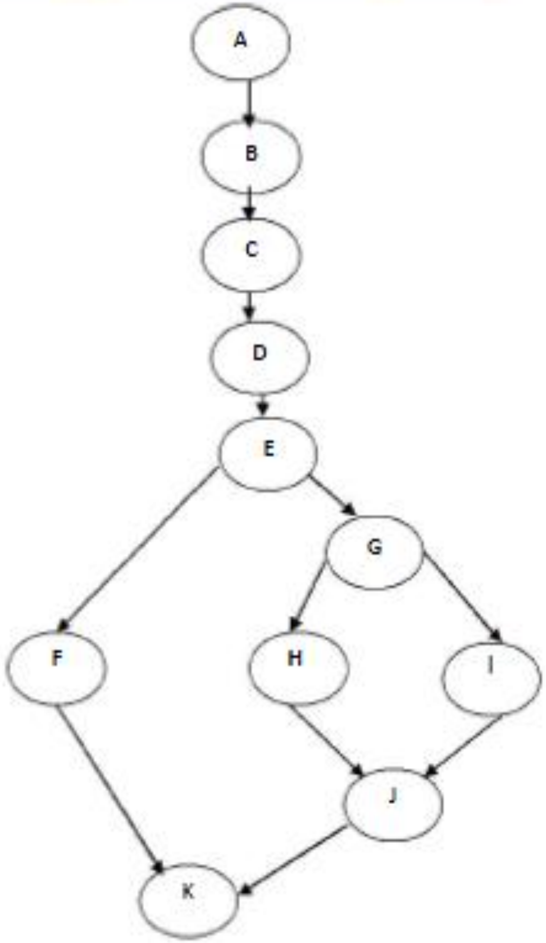


Cont...

DD-paths and Nodes for above example

DD-paths	Nodes
A	7,8,9,10,11,12,13,
B	14
C	15,16,17,18,19,20
D	21,22,23,24,25,26,27,28
E	29
F	30,31,32,33
G	34
H	35,36,37
I	38,39
J	40
K	41,42,42

DD-Path graph of the commission problem



du-Paths for Stocks:

First, let us look at a simple path:the du-path or the variable stocks.Wehae DEF(stocks,15) andUSE(stocks,17),so the path<15,17> is adu-path with respect to stocks. No other is defining nodes are used for stocks; therefore, this path also definition clear.

Cont…

du-Paths for Locks:

Two defining and two usage nodes make the locks variable more interesting: we have DEF(locks,13),
DEF(locks,19),USE(locks,14),and USE(locks,16). These yield four du-paths:

P1=<13,14>
P1=<13,14,15,16>
P1=<19,20,14>
P1=<19,20,14,15,16>
Du-paths p1 and p2 refer to the priming

Value of locks, which is read at node 13: locks has a predicate use in the while statement(node 14), and if the condition is true (as in path p2), acomputation use at statement 16.The other two du-paths start near the end of the while loop and occur when the loop repeats.

du-Paths for totalLocks:

The du-paths for totallocks will be lead us to typical test cases for computations. With two defining nodes(DEF(toalLocks, 10) and DEF(totallocks, 16)) and three usage nodes (USE(totalLocks,16),USE(totalLocks,21), USE(totalLocks,24)), We might expect six du-paths. Let us take a closer look. Path p5=<10,11,12,13,14,15,16> is a du-path in which the initial value of totalLocks(0) has computation use.

du-Paths for Sales:-

Only one defining node is used for sales; therefore, all the du-paths with respect to sales must be definition-clear. They are interesting because they illustrate predicate and computation uses. The

First 3 du-paths are easy:

P10=<27,28>
P11=<27,28,29>
Notice that p12 is a definition-clear path with 3 usage nodes; it also contain paths p10 and p11.
If we were testing withp12, we
know P12=<27, 28,29,30,31,32,33>

We would also cover the other 2 paths.

Cont…

Table 1: Define/Use Nodes for variables in the commission problem

Variable	Defined at Node	Used at Node
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
lockSales	24	27
stockSales	25	27
barrelSales	26	27
sales	27	28, 29, 33, 34, 37, 38
commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

Table 2: Selected define/Use paths

Variable	Path (beginning, end) Nodes	Definition Clear?
lockPrice	7, 24	yes
stockPrice	8, 25	yes
barrelPrice	9, 26	yes
totalStocks	11, 17	yes
totalStocks	11, 22	no
totalStocks	17, 25	no
totalStocks	17, 17	yes
totalStocks	17, 22	no
totalStocks	17, 25	no
locks	13, 14	yes
locks	19, 14	yes
locks	13, 16	yes
locks	19, 16	yes
sales	27, 28	yes
sales	27, 29	yes
sales	27, 33	yes
sales	27, 34	yes
sales	27, 37	yes
sales	27, 38	yes

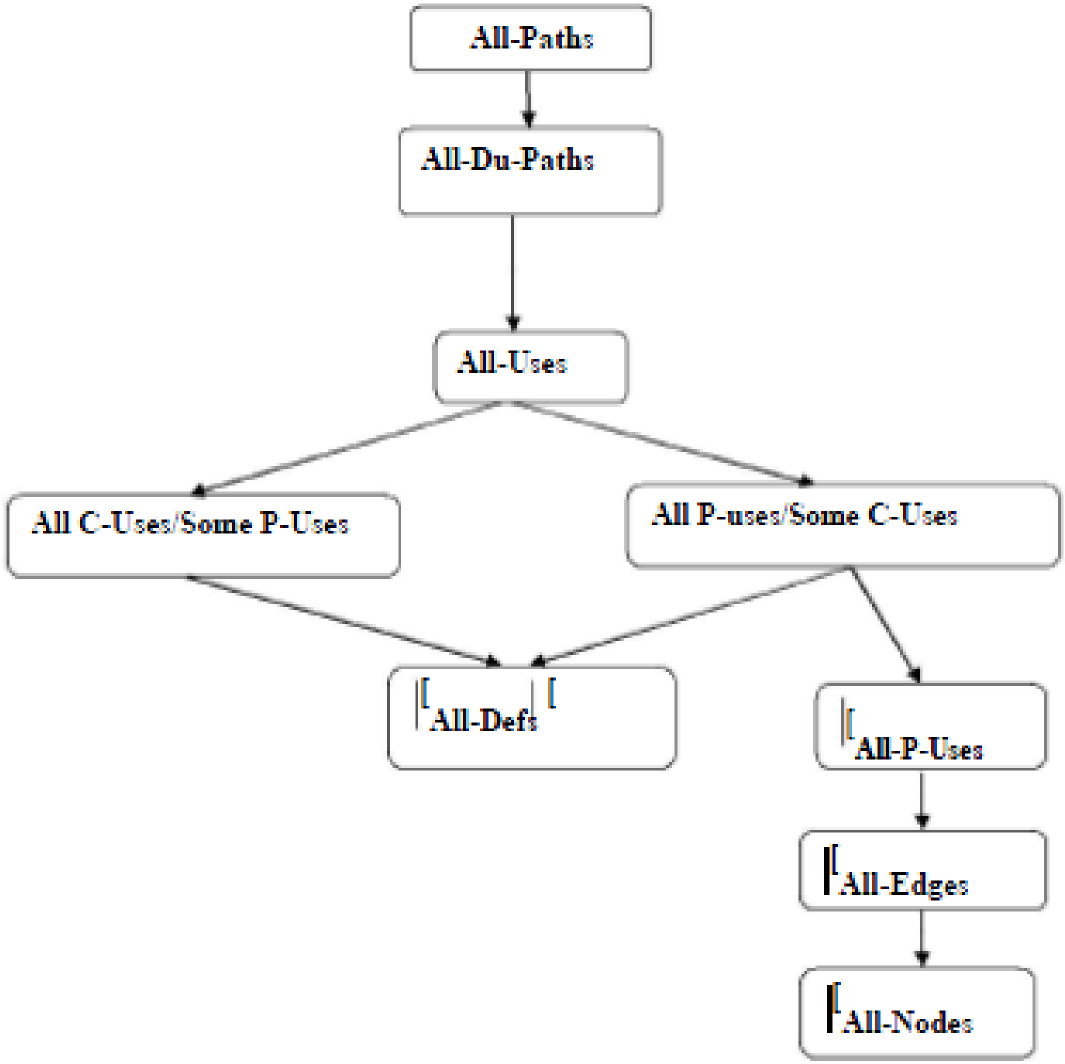
Cont..

Table 3: du-Paths for Commission:

Variable	Path (beginning, end) Nodes	Feasible?	Definition Clear?
commission	31, 32	yes	yes
commission	31, 33	yes	no
commission	31, 37	no	n/a
commission	31, 41	yes	no
commission	32, 32	yes	yes
commission	32, 33	yes	yes
commission	32, 37	no	n/a
commission	32, 41	yes	no
commission	33, 32	no	n/a
commission	33, 33	yes	yes
commission	33, 37	no	n/a
commission	33, 41	yes	yes
commission	36, 32	no	n/a
commission	36, 33	no	n/a
commission	36, 37	yes	yes
commission	36, 41	yes	no
commission	37, 32	no	n/a
commission	37, 33	no	n/a
commission	37, 37	yes	yes
commission	37, 41	yes	yes
commission	38, 32	no	n/a
commission	38, 33	no	n/a
commission	38, 37	no	n/a
commission	38, 41	yes	yes

Test Coverage Metrics

du-path Test Coverage Metics



SLICE-BASED TESTING

Data flow testing focuses on the points at which variables **receive values** and the points at which these values are **used (or referenced). It detects improper use** of data values (data flow anomalies) due to coding errors.

Rapps and Weyukers Motivation*: " it is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, **one should not feel confident** about a program without having seen the effect of Using the value produced by each and every computation.

- Data flow testing
  1. Define / Use Testing
  2. Slice-Based Testing

Slice-Based Testing

The following refers to program P that has program graph G (P) and the set of program variables V. In a program graph statement fragments are nodes and edges represents node sequence .G (P) has single entry and single exit node. We also disallow edges from node to itself. The set of all paths in P is PATHS (P)

Definition:-Given a program P and a set V of variables in P, a slice on the variable set V at statement n, written S(V,n), is the set of all statements in P prior to node n that contribute to the values of variables in V at node n. Listing elements of a slice S(V, n) will be cumbersome because the elements are program statement fragments. It is much simpler to list fragment numbers in P(G).

Cont…

USE TYPES	DEF TYPES
P-use - Used in a predicate stmt	I-def-Defined by input
C-use - Used in computation	A-def-Defined by assignment
O-use -Used for output	
L-use - used for location (pointers)	
I-use Iteration (Internal counters, loop indices)	

Commission Problem

Example :- The commission problem is used here because it contains interesting dataflow properties , and these are not present in the triangle problem(or in next date function).Follow these examples while looking at the source code for the commission problem that we used to analyse in terms of define/use paths.

(Commission problem)

```
1. program commission(INPUT, OUTPUT)
2. Dim lock , stock , barrels as Integer
3. Dim lockprice ,stockprice , barrelprice As Real
4. Dim totalLocks ,totalStocks , totalBarrels As Integer
5. Dim lockSales, stocksales ,barrelsSales As Real
6. Dim sales , commission As Real
7.  lockprice=45.0
8 stockprice= 30.0
9.  barrelprice = 25.0
10. totallocks=0
11. totalstocks=0
12.  totalbarrels=0
13 .input(locks)
14.Whle not (locks= -1)
15.Input (stock,barrel)
16.Totallocks =total locks +locks
17. Totalstocks =total stocks +stocks
18. Totalbarrels = totalbarrels +barrels
```

Cont…

19. input (locks)

20.End While

21. output ("Locks sold", total locks)

22. output ("Stocks sold", total stocks)

23. output ("Barrels sold", totalbarrels)

24.locksales= lockprice *totallocks

25.stocksales= stockprice *totalstocks

26.barrelsales= barrelprice *totalbarrels

27 sales= locksales + stocksales+barrelsales

28. output("Totalsales", sales)

29. if (sales > 1800.0)

30. then

31. commission = 0.10 * 1000

32. commission =commission +0.15 *800.0

33.commission = commission +0.20 *(sales >1000)

34. Else if (sales >1000)

35.Then

36. commission = 0.10 * 1000.0

37. commission = commission +0.15 *(sales-1000.0)

38. Else

39. commission =0.10 * sales

40. End If

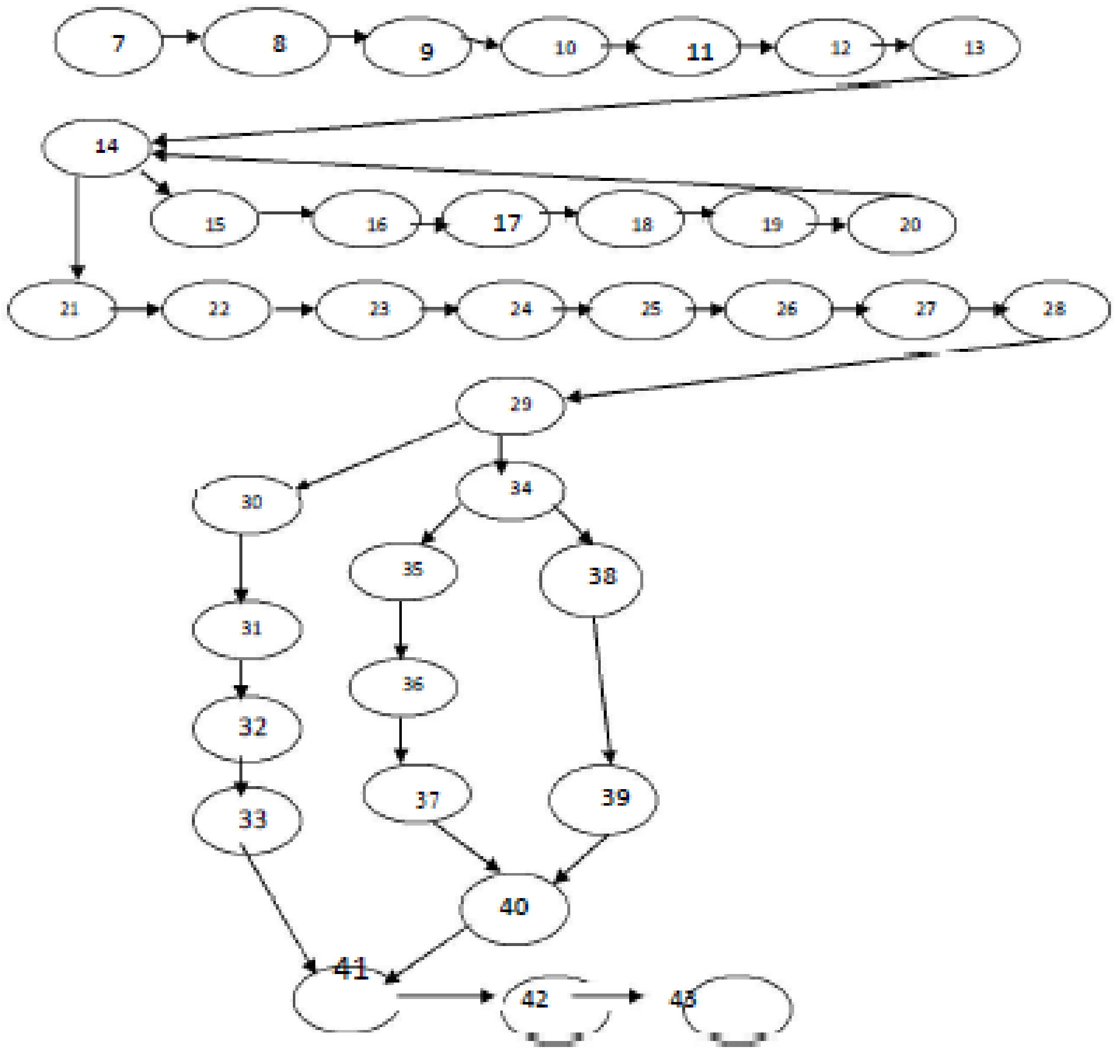
41. End If

42. output ("commission is $", commission)

43. End commission.

Cont…

Program graph of the the commission problem



Cont…

Slices on the locks variable show why it is potentially fault-prone.it has a P-use at node 14 and a C-use at node 16 and has two definitions, the I-defs at nodes 13 and 19.

S1:S(locks,13)={13}
S2:S(locks, 14)={13,14,19,20}
S3:S(locks,16)={13,14,19,20}
S4:S(locks,19)={19}

The Slices for stocks and barrels are boring. They are short, definition-clear paths contained Entirely within a loop, so they are not affected by iterations of the loop. (Think of the loop body
As a DD-Path.)

S5:S(stocks,15)={13,14,15,19,20}
S6:S(stocks,17)={13,14,15,19,20}
S7:S(barrels,15)={13,14,15,19,20}
S8:S(barrels,18)={13,14,15,19,20}

The next three slices illustrates how repetation appears in slices. Node 10 is an A-def for totalLocks
And node 16 contains both an A-def and a C-use. The remaining nodes in S10(13, 14,19 and 20) pertain to the While loop controlled by locks. Slice S10 and S11 are equal because nodes 21 and 24 are an O-use and a C-use of totalLocks respectively.

S9:S(totalLocks,10)={10}
S10:S(totalLocks,16)={10,13,14,16,19,20}
S11:S(totalLocks,21)={10,13,14,16,19,20}

The slices on totalStocks and totalBarrels are quite similar. They are initialized by A-defs at nodes 11 and 12 and then are defined by A-defs at nodes 17 and 18. Again, the remaining nodes (13,14,19 and 20) pertains to the While loop controlled by locks.

S12:S(totalLocks,11)={11}
S13:S(totalLocks,17)={11,13,14,15,17,19,20}
S14:S(totalLocks,22)={11,13,14,15,17,19,20}
S15:S(totalBarrels,12)={12}
S16:S(totalBarrels,18)={12,13,14,15,16,19,20}
S17:S(totalBarrels,23)={12,13,14,15,18,19,20}

Test Execution

It is the process of executing test cases intended to find defects.

Automating Test Execution

- Designing test cases and test suites is creative
  - Like any design activity: A demanding intellectual activity, requiring human judgment
- Executing test cases should be automatic
  - Design once, execute many times
- Test automation separates the creative human process from the mechanical process of test execution

Scaffolding (A Temporary Structure)

Code developed to facilitate testing is called **scaffolding**

Scaffolding has different parts

1. Test Harnesses
2. Drivers
3. Stubs

Scaffolding was made popular by the Ruby on Rails framework.
It has been adapted to other software frameworks, including **OutSystems Platform,Expressframework, Playframework, Django, MonoRail, Brail, Symfony, Laravel, CodeIgniter, Yii, CakePHP, Phalcon PHP, Model-Glue, PRADO, Grails, Catalyst, Seam Framework, Spring Roo, ASP.NET Dynamic Data and ASP.NET MVC framework‘s**..etc

Scaffolding ...

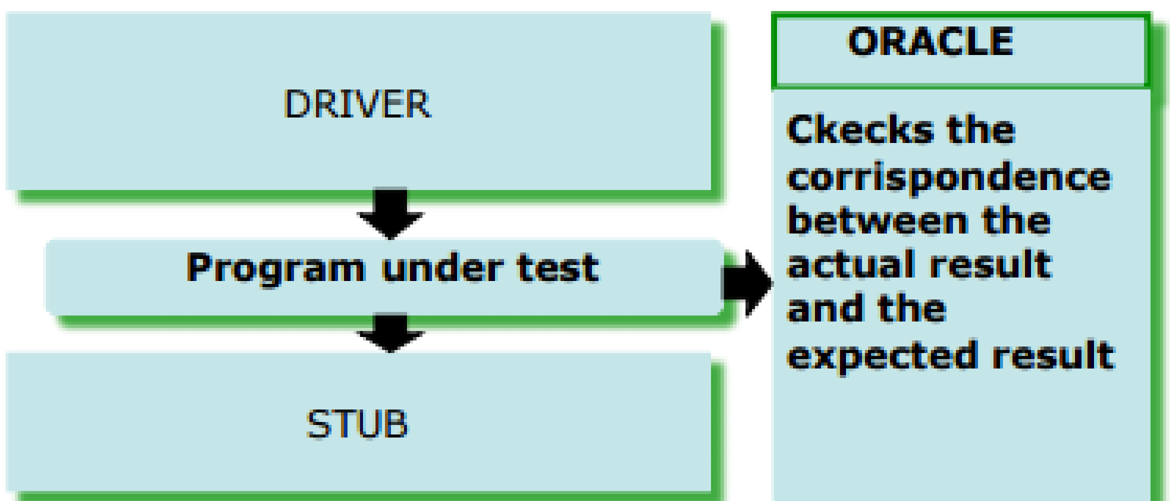
- Test driver
  - A "main" program for running a test
    - May be produced before a "real" main program
    - Provides more control than the "real" main program
      - To driver program under test through test cases
- Test stubs
  - Substitute for called functions/methods/objects
- Test harness
  - Substitutes for other parts of the deployed environment
    - Ex: Software simulation of a hardware device

Test Oracles

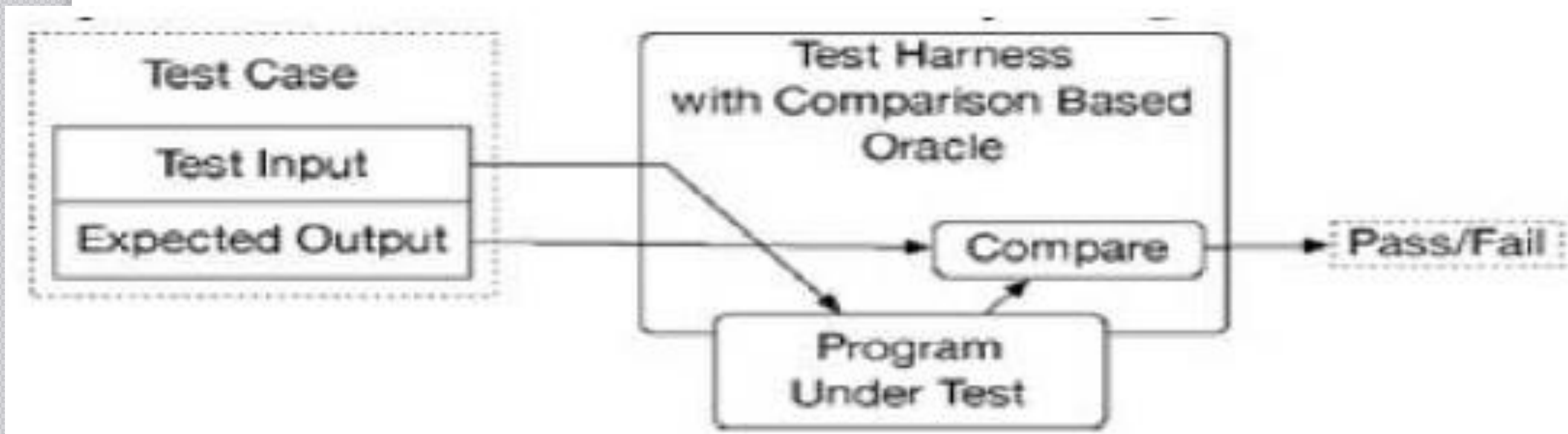
Software that applies a pass/fail criterion to a program execution is called a test oracle, often called as oracle

Oracles

An essential part of the test scaffolding



Cont…


Test Case
Test Input
Expected Output
Test Harness
with Comparison Based
Oracle
Compare
Pass/Fail
Program
Under Test

Capture and Replay

- Sometimes there is no alternative to human input and observation
  - Even if we separate testing program functionality from GUI, some testing of the GUI is required
- We can at least cut *repetition* of human testing
- *Capture* a manually run test case, *replay* it automatically
  - with a comparison-based test oracle: behavior same as previously accepted behavior
    - reusable only until a program change invalidates it
    - lifetime depends on abstraction level of input and output

Capture and Replay Tools

- Often used for regression test development
  - Tool used to capture interactions with the system under test.
  - Inputs must be captured; outputs may also be recorded and (possibly) checked.
  - Examples:
    - GUI testing tools
- Capture requires a working system to be available already!

Content of The capture record

- Inputs, outputs, and other information needed to reproduce a session with the system under test need to be recorded during the capture process.
- Examples:
  - General information: date/time of recording, etc.
  - System start-up information
  - Events from test tool to system
    - Point of control, event
  - Events from system to test tool
    - Checkpoints / expected outputs
  - Time stamps

Integrating a Capture and Replay tool

- GUI frameworks are typically event-driven architectures
  - Various controls create events when they are created, activated, modified, deactivated, or disposed.
  - Input devices create events as per their functions: key pressed, key released, mouse moved, ...
  - Events are sent to an event dispatcher
- During the capture process, the tool will register as an event listener
  - Event notification method for the tool will record the details of all events that occurred.
- During the replay process, the tool will register as an event source (possibly also as a listener)
  - For mouse and keyboard events, the tool has to substitute for the actual devices as the event source.
    - Replay events should be initiated at the same relative time as during the capture.
  - Other controls issue events as usual (e.g. GUI button deactivated)

Conclusion

In a nut shell we have seen a brief Introduction to Structural Testing, Test Execution, Scaffolding, Test Oracles, Capture & Reply.

Software Testing

MODULE-4:PROCESS FRAMEWORK

Agenda

1. Validation
2. Verification
3. Relationship Between Validation & Verification
4. Dependability
5. Difference between validation & Verification
6. Degree of freedom
7. Basic Principles of Analysis & Testing
8. Improving the process
9. Conclusion

Process Framework

Process:

Process is a series of actions or steps taken in order to achieve a particular end.

Framework:

A framework is often a layered structure indicating what kind of programs can or should be built and how they would interrelate.

Process framework deals with the different steps in a procedural manner , here we design test framework in terms of process setup in the testing Team.

What Is Validation?

- Assessing the degree to which a software system actually fulfills its requirements, in the sense of meeting the user's real needs, is called validation.
- Are we building the right product???

What Is Verification?

- Checking the consistency of an implementation with a specification.
- An overall design could play the role of “specification”.
- A more detailed design could play the role of “Implementation”.
- Are we building the product right????

Difference between software **Verification and Validation**

Verification	**Validation**
Are we building the system right?	Are we building the right system?
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	Validation is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.	The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
Following activities are involved in Verification: Reviews, Meetings and Inspections.	Following activities are involved in Validation: Testing like black box testing, white box testing, gray box testing etc.
Verification is carried out by SQA team to check whether implementation software is as per specification document or not.	Validation is carried out by testing team.
Execution of code is not comes under Verification.	Execution of code is comes under Validation.
Verification process explains whether the outputs are according to inputs or not.	Validation process describes whether the software is accepted by the user or not.
Verification is carried out before the Validation.	Validation activity is carried out just after the Verification.
Following items are evaluated during Verification: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,	Following item is evaluated during Validation: Actual product or Software under test.
Cost of errors caught in Verification is less than errors found in Validation.	Cost of errors caught in Validation is more than errors found in Verification.
It is basically manually checking the of documents and files like requirement specifications etc.	It is basically checking of developed program based on the requirement specifications documents & files.

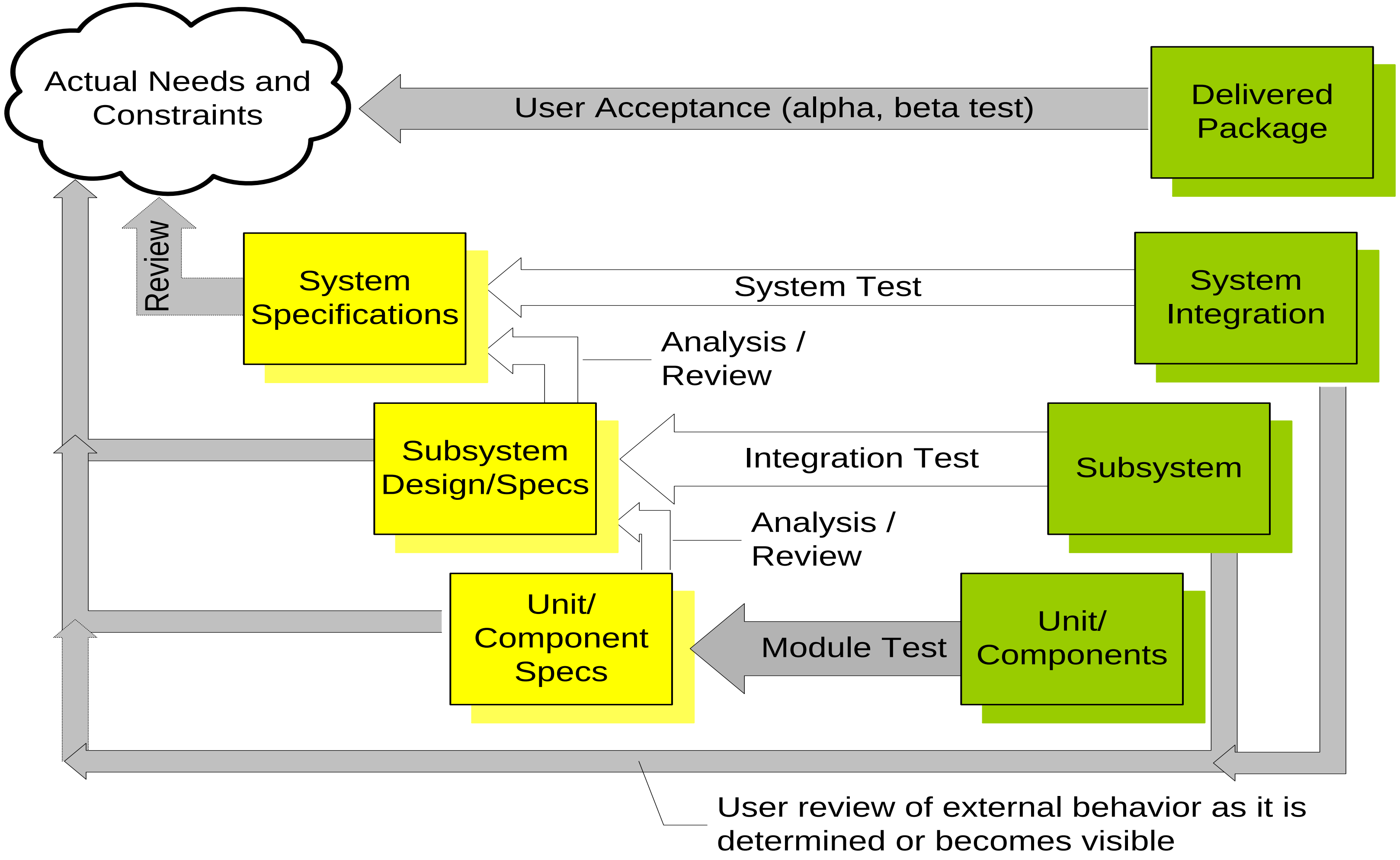
Conclusion on difference of ***Verification and Validation in software testing***

- Both Verification and Validation are essential and balancing to each other.
- Different error filters are provided by each of them.
- Both are used to finds a defect in different way, Verification is used to identify the errors in requirement specifications & validation is used to find the defects in the implemented Software application.

Agenda

1. Validation
2. Verification
3. **Relationship Between Validation & Verification**
4. Dependability properties
5. Degree of freedom
6. Basic Principles of Analysis & Testing
7. Improving the process
8. Conclusion

Relationship of verification & Validation



The Relation Of Verification And Validation Activities With Respect To Artifacts Produced In a Software Development Project

Cont...

- Verification Activities Checks Consistency B/W Designs And Specifications At Adjacent Level.

- Validation Activities Attempts To Guage Whether The System Actually Satisfies Its Intended Purpose.

- Validation Activities Refer Primarily To Overall System Specification And The Final Code.

- Overall System Specification → Discrepancies B/W Actual Needs And System Specification.

Cont...

- Final Code→ Discrepancies B/W Actual Product and the final product.
- Verification includes checks for self-consistency and well-formedness.
- Ex: we cannot judge that a program is “correct” except in Reference to a specification of what it should do, we can certainly determine that some programs are “Incorrect” because they are Ill-formed.

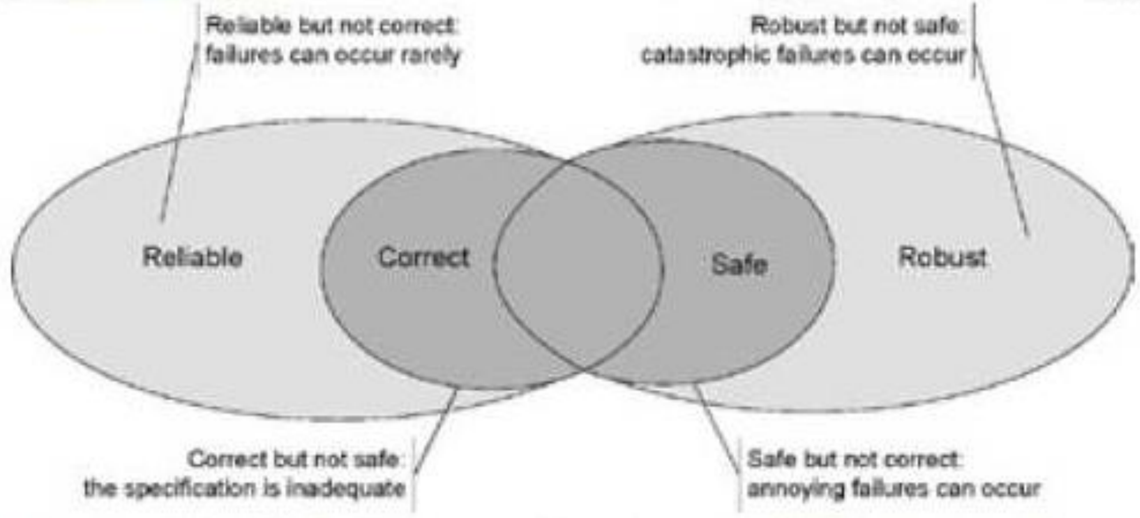
Agenda

1. Validation
2. Verification
3. Relationship Between Validation & Verification
4. **Dependability Properties**
5. Degree of freedom
6. Basic Principles of Analysis & Testing
7. Improving the process
8. Conclusion

Dependability Properties

1. Reliability
2. Correctness
3. Safety
4. Robustness

illustrates the relation among dependability properties.



Cont…

- **Correctness** → Absolute Consistency With Specification.
- **Reliability** → Correct behaviour In Expected Use.
- **Robustness** → Behaviour Under Exceptional Conditions.
- **Safety** → Avoidance of Particular Hazards.

Agenda

1. Validation
2. Verification
3. Relationship Between Validation & Verification
4. Dependability Properties
5. **Degree of freedom**
6. Basic Principles of Analysis & Testing
7. Improving the process
8. Conclusion

Degrees Of Freedom-Definition

- Measure of how many values can vary in a statistical calculation
- There must exist a logical proof that a program satisfies all its specifications
- Easy to obtain such proofs for simple programs though at high cost
- **In general, One can't produce a completely, logically correct proof that a program will work in all systems & at all inputs**

Undecidability Theory

- For each verification technique checking a property "S", at least one pathological program exists for which a correct answer will never be obtained in finite time.
- Verification will fail at least in one case.
- i.e. significant degree of inaccuracy must be accepted

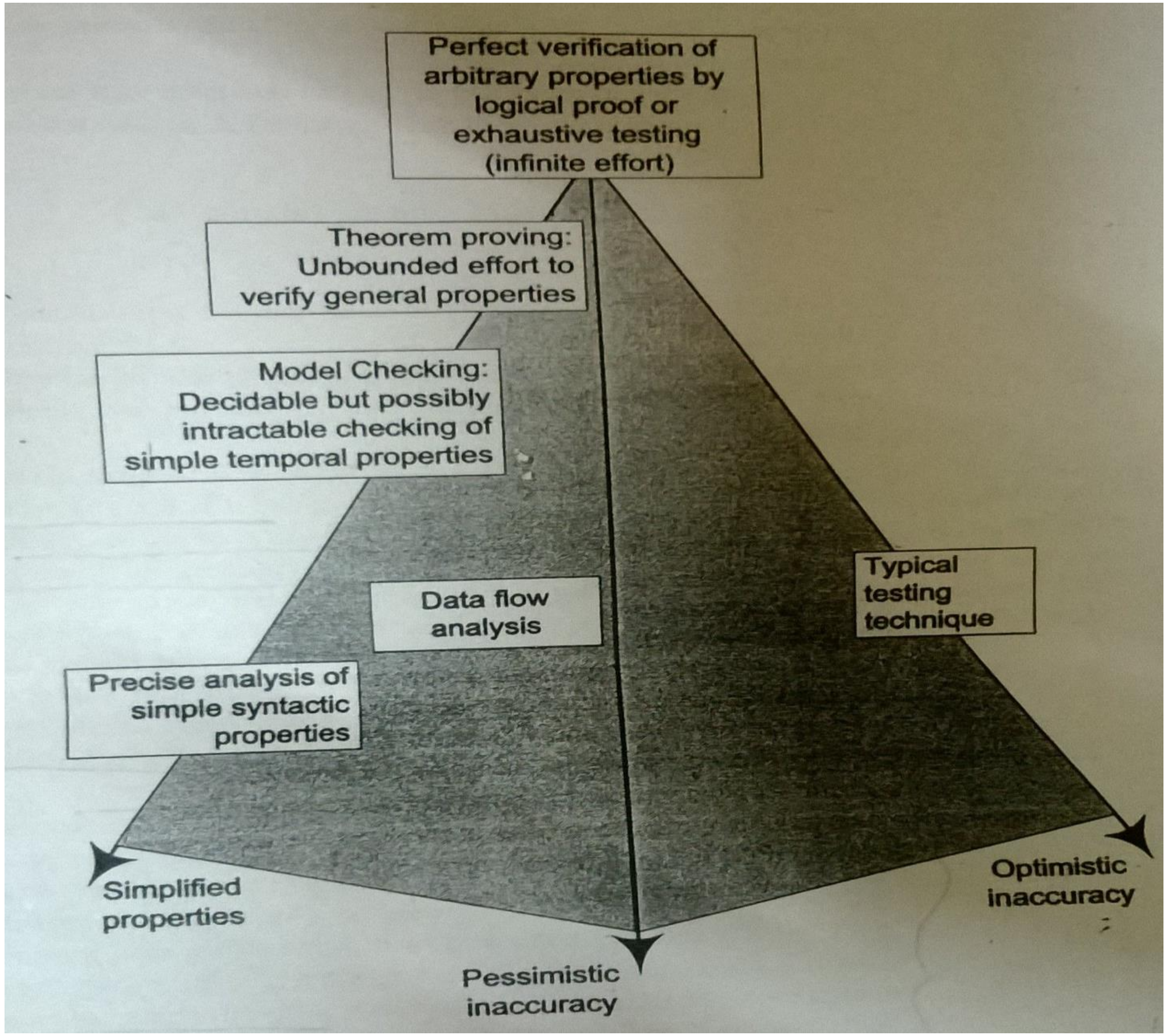
Need for Logical Proof

Consider the following cases:

```
   class A
{
    static int sum(int a, int b)
{
   return a+b;
}
}
```

- Its an example of a java class
- Representation of int is 32 binary digits
- 2^32 x 2^32 = 2^64 = 10^21 different inputs on which A sum() has to be tested for correctness proof
- At 1 ns(10^-9 secs) per test case which will take about 30,000 years

Verification trade-off dimensions



Inaccuracies in verification technique

Pessimistic inaccuracy

- The failure to accept even correct programs
- Not guaranteed to accept a program even if it possess the specified properties

Optimistic Inaccuracy

- Failure to reject incorrect programs
- Accepts programs that do not posses specified properties
- Doesn't detect all violations to the specifications

Conservative analysis

- Verification technique that follows pessimistic approach

Drawbacks

- Produces large number of spurious error reports with a few accurate report
- Programmer will be unable to deal with a long list of mostly false alarms

✓ Since perfection is unobtainable, we must choose a technique that acts as an intermediate between pessimistic & optimistic Inaccuracy

Introducing simple checks

Program

```
Int i, sum;
Int first=1;
For(i=0;i<10,++i)
{
If (first)
{
Sum=0; first=0;
}
Sum+=I;
}
```

- ✓ **Rule:** each variable should be initialized before its value is used in any expression
- ✓ **Java solved this problem by making such code illegal**

Agenda

1. Validation
2. Verification
3. Relationship Between Validation & Verification
4. Dependability Properties
5. Degree of freedom
6. **Basic Principles of Analysis & Testing**
7. Improving the process
8. Conclusion

Basic Principles of Analysis & Testing

As in any engineering discipline, techniques of analysis and testing software follow few key principles.

Different Principles are given below: **[SRRPVF]**

1. Sensitivity
2. Redundancy
3. Restriction
4. Partition
5. Visibility
6. Feedback

Sensitivity

1. Better to fail every time than sometimes
2. Sensitivity requires techniques of abstraction: system behavior cannot be related to specific circumstances .

When it uses a systematic strategy (e.g. using checklists or guidelines), **code inspection can help to find faults on regular basis.**

Code Inspection

✓Inspection is a peer review process operated by trained individuals who look for defects.

✓A Fagan inspection is a structured inspection process which includes inspection planning, overview meeting, preparation, inspection meeting, rework, follow-up

✓Code review is an inspection to discover bugs in a particular piece of code.

✓Code review is more informal, tool-based, and used regularly in practice than Fagan

Redundancy

✓From information theory: redundancy means dependency between transmissions.

✓Solution: create guards against transmission errors .

✓In software, redundancy means **consistency between intended and actual system behavior**.

✓Solution: create guards for artifacts consistency, making intention explicit. [RTM]

Ex:

- ✓Redundancy as dependency among parts of code by using a variable:
- ✓a variable is defined and then used elsewhere.
- ✓Type declaration is a technique that makes the **intention explicitly**.
- ✓Type declaration constraints the variable use in **other part of the code**.
- ✓**Compilers check the correct use of a variable against its declared type.**

Restriction

Substituting principle

1. Making the problem easier or
2. Reducing the set of classes under test

Substituting principle

In complex system, verifying properties can be infeasible. Often this happens when properties are related to specific human judgements, but not only substituting a property with one that can be easier verified or constraining the class of programs to verify

- Separate human judgment from objective verification.
- Example: Property: Each ”relevant” term in the dictionary must have a definition in the glossary. Separate the term ”relevant” giving it a standard for example.
- Example: “Race condition”: interference between writing data in one process and reading or writing related data in another process (an array accessed by different threads).

  Testing the integrity of shared data is difficult as it is checked at run time.

  Typical solution is to adhere to a protocol of serialization

Cont…

Example

```
static void questionable(){
int k;
  for(int i=0; i<10;i++){
    if(someCondition(i)){
        k=0;
    }
  }
}
```

Compilers cannot be sure that k will be ever initialized, depends on the condition

Make the problem easier: Java does not allow this code

Partition

Partition

Divide and conquer. Typical engineering principle. There are several ways to apply it in testing, for instance:

- Divide testing into unit, integration, subsystem and system testing to focus on different types of faults at different stages and at each stage take advantage of the result of the previous stage
- Separate the program from one model of it and test a given property on the model

Partition testing divides input into classes of equivalent expected output.

- Then test criteria identify representatives in classes to test a program
- A general rule to identify representatives does not exist otherwise equivalence between programs would be possible

Statement coverage checks whether all statements are executed at least once.

Visibility

✓Setting goals and methods to **achieve those goals**

✓Making information **accessible to the user**

Feedback

Apply lessons learned from **experience in process** improvement and techniques

✓Iterative testing in eXtreme programming

✓Prototyping of the same

Agenda

1. Validation
2. Verification
3. Relationship Between Validation & Verification
4. Dependability Properties
5. Degree of freedom
6. Basic Principles of Analysis & Testing
7. **Improving the process**
8. Conclusion

Why Improvement in Process?

- Commonality of projects undertaken by an **organization over time**.
- Developers tend to make the **same kind of errors, over and over due to which same kinds of software faults are encountered.**
- Quality process can be improved by **gathering, analyzing and acting on data** regarding faults and failures.

How To Do It?

- Gather sufficiently **complete and accurate data about faults and failures**.
- Integrate data collection with **other development activities**.
- E.g.:- **Version and configuration control, project management and bug tracking.**
- Minimize extra effort.
- Aggregate raw data on faults and failures into categories and prioritize them.

Analysis Step

- Tracing several instances of an observed fault and failure, back to the human error from which it resulted.
- Involves the reasons as to why the faults were not detected and removed earlier.- "Root Cause Analysis"
- Counter measures involve changing the
  1. Programming methods or
  2. Improvements to quality assurance activities or
  3. Change in management practices.

Organizational Factors

- Poor allocation of responsibilities can lead to major problems in which pursuit of individual goals conflicts with overall project success.
- Different teams for development and quality?
  - separate development and quality teams is common in large organizations
  - indistinguishable roles is postulated by some methodologies (extreme programming)
- Different roles for development and quality?
  - Test designer is a specific role in many organizations
  - Mobility of people and roles by rotating engineers over development and testing tasks among different projects is a possible option

Agenda

1. Validation
2. Verification
3. Relationship Between Validation & Verification
4. Dependability Properties
5. Degree of freedom
6. Basic Principles of Analysis & Testing
7. Improving the process
8. **Conclusion**

CONCLUSION

In a nut shell, we have seen definition of Validation, Verification, Relationship Between Validation & Verification, Dependability, Difference between validation & Verification, Degree of freedom, Undecidability Theory, Need for logical Proof, Pessimistic & Optimistic Inaccuracies, Basic Principles of Analysis & Testing and Improving the process

Planning and Monitoring the Process, Documenting Analysis and Test

Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Planning and Monitoring

What are Planning and Monitoring?

- Planning:

– Scheduling activities (what steps? in what order?)
– Allocating resources (who will do it?)
– Devising unambiguous milestones for monitoring

- Monitoring:

Judging progress against the plan
– How are we doing? -- Red, Amber and Green

- A good plan must have ***visibility*** :

– Ability to monitor each step, and to make objective judgments of progress

Agenda

- ✓Planning and Monitoring
- ✓Quality and Process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Quality and Process

Quality Process:

Set of activities and responsibilities
– focused primarily on ensuring adequate dependability
– concerned with project schedule or with product usability

- **A framework for**

– selecting and arranging activities
– considering interactions and trade-offs

- **Follows the overall software process in which it is embedded**

– Example: waterfall software process —> "V model": unit testing starts with implementation and finishes before integration.
– Example: (Extreme Programming) XP and Agile methods —> emphasis on unit testing and rapid iteration for acceptance testing by customers

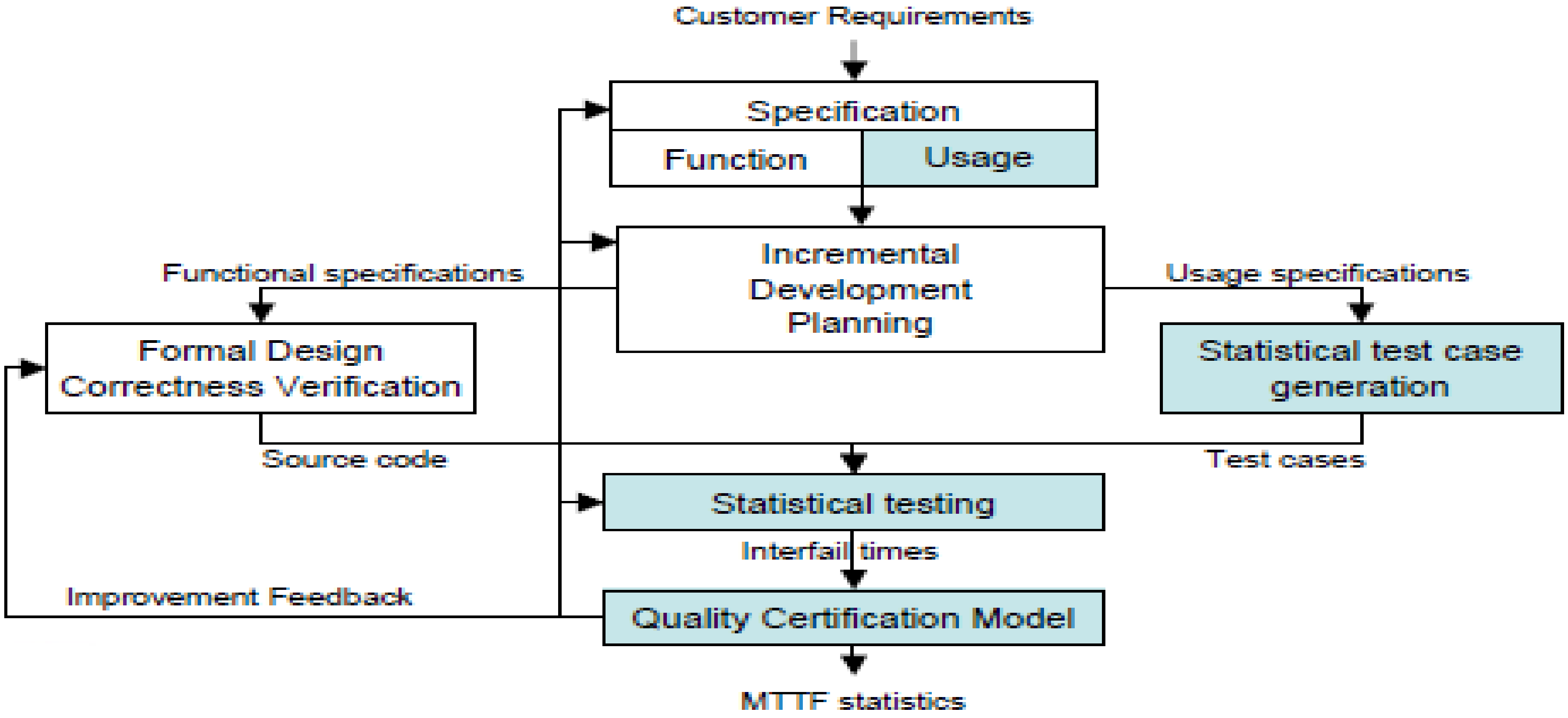
Clean Room Process

✓The cleanroom software engineering process is a software development process intended to produce software with a certifiable level of reliability. (**Software Reliability** is the probability of **failure-free software** operation for a specified period of time in a specified environment.)

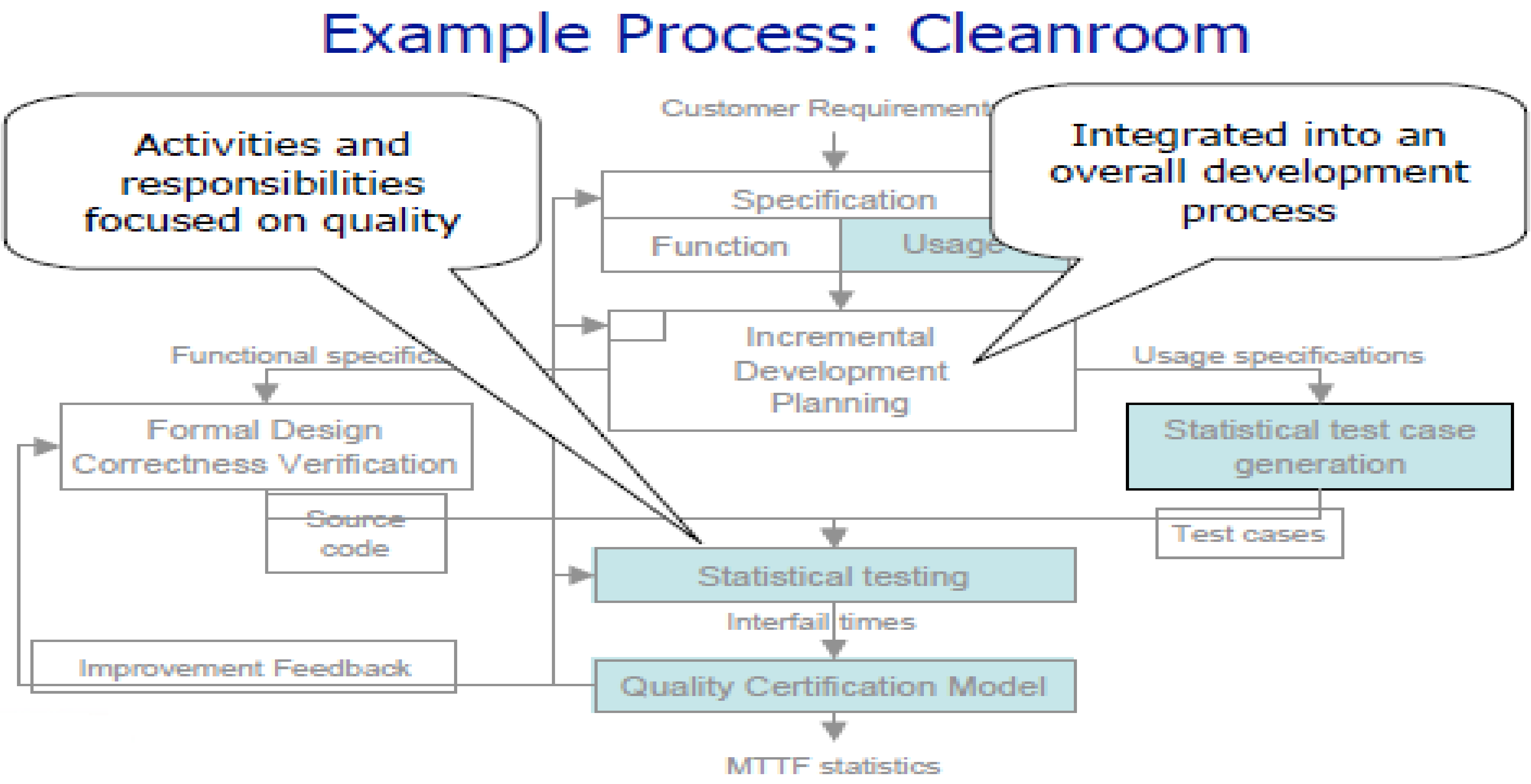
✓The cleanroom process was originally developed by Harlan Mills and several of his colleagues including Alan Hevner at IBM. The focus of the cleanroom process is on **defect prevention, rather than defect removal**.

✓The name "cleanroom" was chosen to invoke the cleanrooms used in the electronics industry to prevent the introduction of defects during the fabrication of semiconductors.

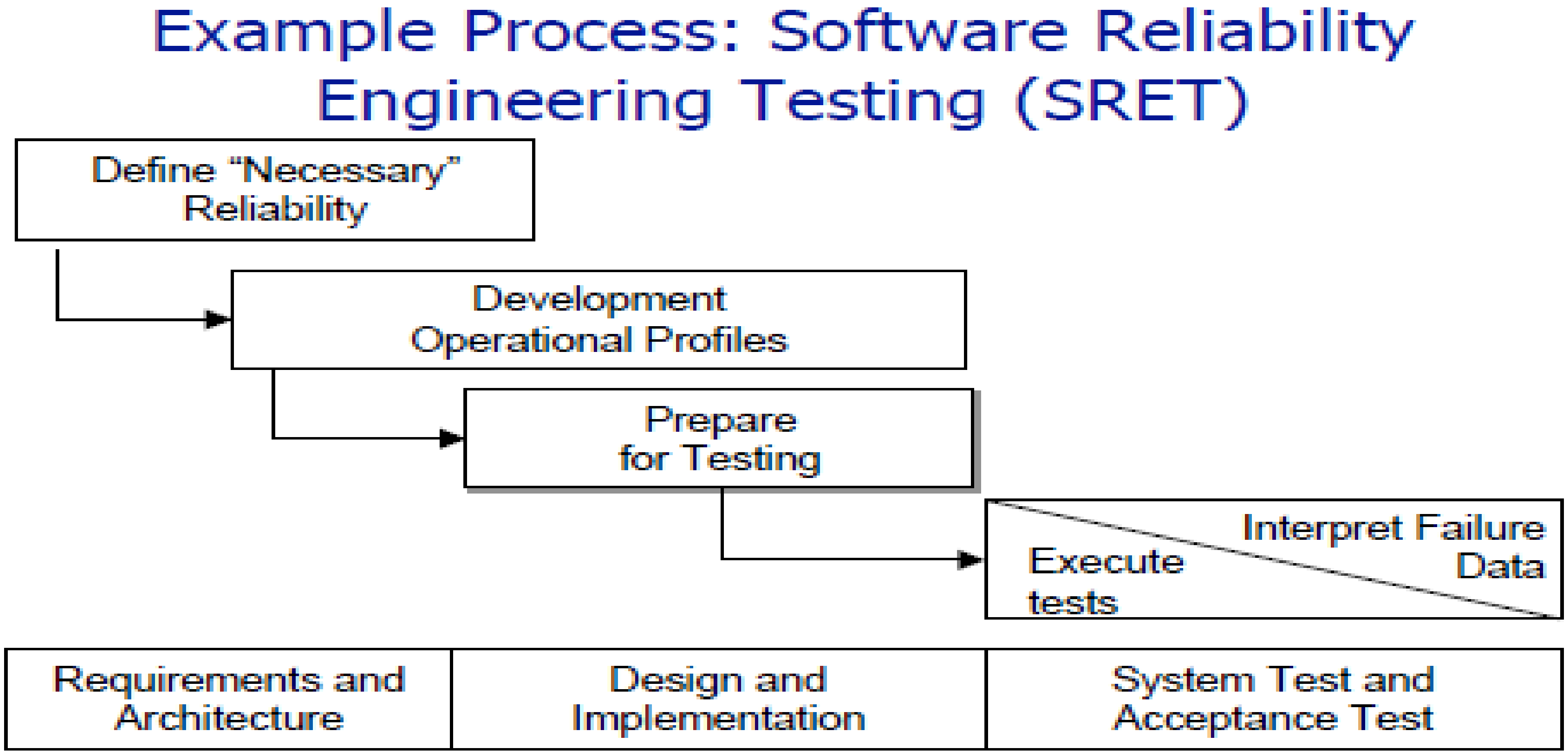
Example Process: Cleanroom



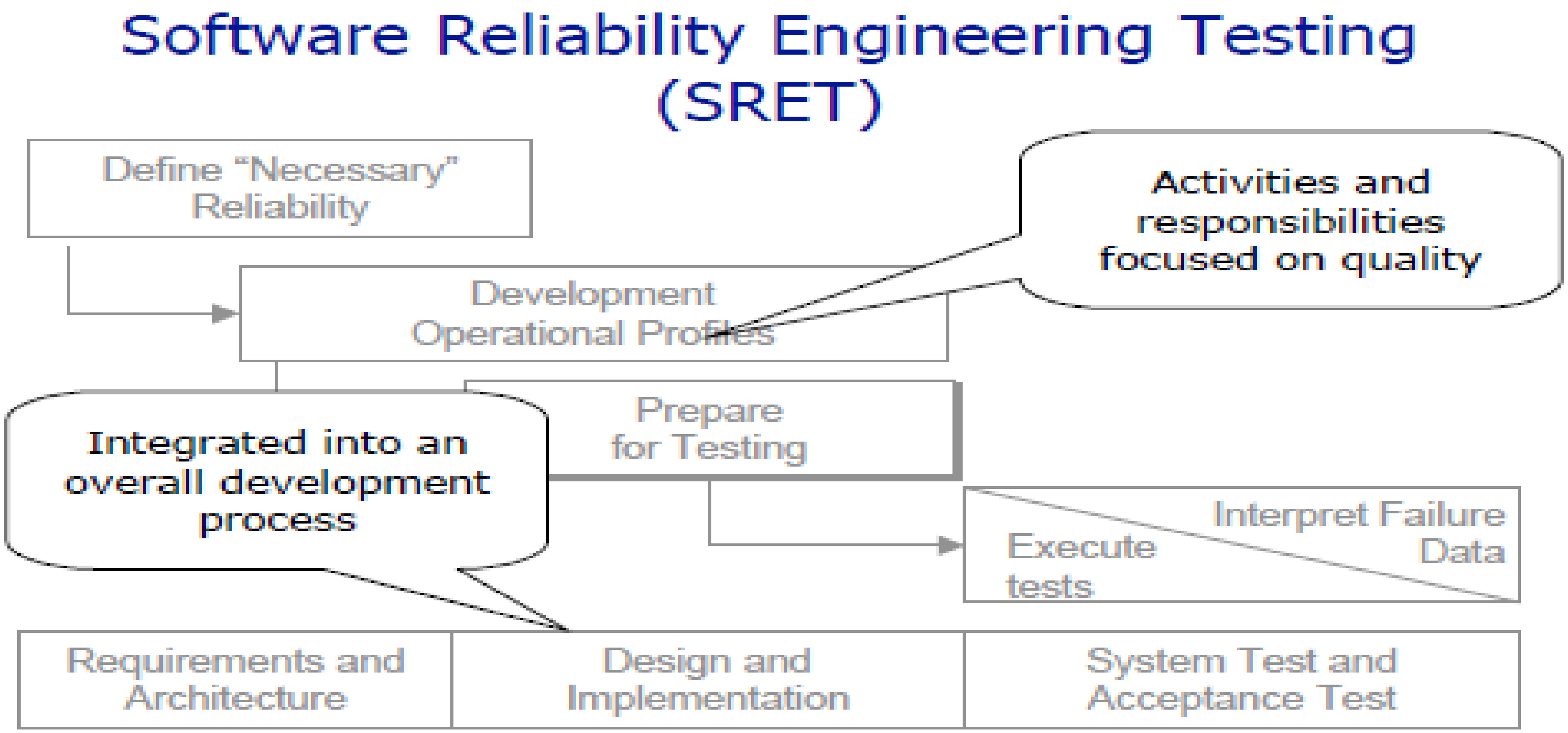
Cont…


Example Process: Cleanroom
Activities and responsibilities focused on quality
Customer Requirements
Specification
Function
Usage
Integrated into an overall development process
Incremental Development Planning
Functional specific
Usage specifications
Formal Design Correctness Verification
Statistical test case generation
Source code
Test cases
Statistical testing
Interfail times
Improvement Feedback
Quality Certification Model
MTTF statistics

Software Reliability Engineering Testing (SERT)



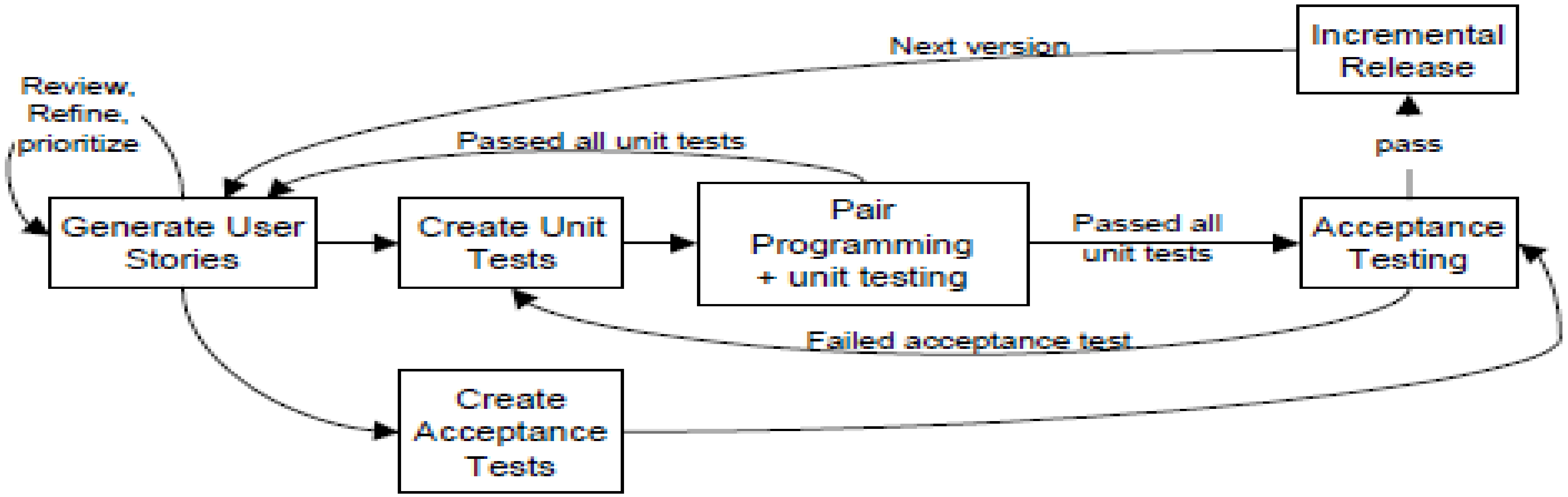
Cont…



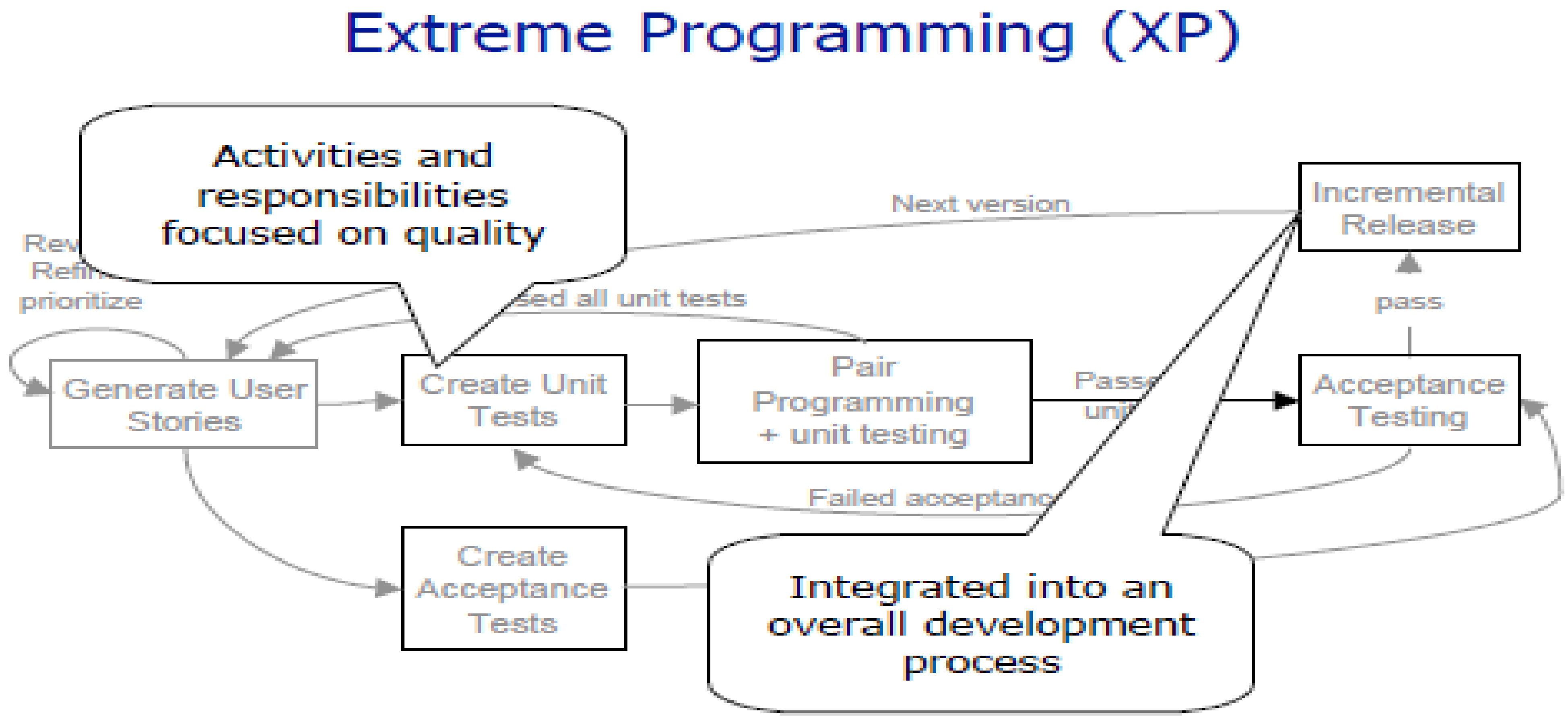
Software Reliability Engineering Testing (SRET)
Define "Necessary" Reliability
Development Operational Profiles
Activities and responsibilities focused on quality
Prepare for Testing
Integrated into an overall development process
Execute tests
Interpret Failure Data
Requirements and Architecture
Design and Implementation
System Test and Acceptance Test

Extreme Programming

Example Process: Extreme Programming (XP)



Cont...



Extreme Programming (XP)
Activities and responsibilities focused on quality
Rev
Refin
prioritize
Next version
Incremental Release
sed all unit tests
pass
Generate User Stories
Create Unit Tests
Pair Programming + unit testing
Acceptance Testing
Failed acceptanc
Create Acceptance Tests
Integrated into an overall development process

Overall Organization of a Quality Process

Key principle of quality planning

– The cost of detecting and repairing a fault increases as a function of time between committing an error and detecting the resultant faults.

- therefore ...

– An efficient quality plan includes matched sets of intermediate validation and verification activities that detect most faults within a short time of their Introduction.

- and ...

– V&V steps depend on the intermediate work products and on their anticipated defects.

Verification Steps for Intermediate Artifacts

- Internal consistency checks

– Compliance with structuring rules that define "well-formed" artifacts of that type
– A point of leverage: define syntactic and semantic rules thoroughly and precisely enough that many common errors result in detectable violations.

- External consistency checks

– Consistency with related artifacts
– Often: conformance to a "prior" or "higher-level" specification

- Generation of correctness conjectures (Inferences)

– Correctness conjectures: lay the groundwork for external consistency checks of other work products
– Often: motivate refinement of the current product

Strategies vs Plans

	Strategy	*Plan*
Scope	Organization	Project
Structure and content based on	Organization structure, experience and policy over several projects	Standard structure prescribed in strategy
Evolves	Slowly, with organization and policy changes	Quickly, adapting to project needs

Agenda

- ✓Planning and Monitoring
- ✓Quality and Process
- ✓Test and Analysis Strategies and Plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Test and Analysis Strategy

- Lessons of past experience

– An organizational asset built and refined over time

- Body of explicit knowledge

– More valuable than islands of individual competence
– Amenable (Agreeable) to improvement
– Reduces vulnerability to organizational change (e.g., loss of key individuals)

- Essential for

– Avoiding recurring errors
– Maintaining consistency of the process
– Increasing development efficiency

Considerations in Fitting a Strategy to an Organization

•Structure and size

– example

• Distinct quality groups in large organizations, overlapping of roles in smaller organizations

• greater reliance on documents in large than small organizations

• Overall process

– example

• Cleanroom requires statistical testing and forbids unit testing

– fits with tight, formal specs and emphasis on reliability

• XP prescribes "test first" and pair programming

– fits with fluid specifications and rapid evolution

• Application domain

– example

• Safety critical domains may impose particular quality objectives and require documentation for certification (e.g,RTCA/DO-178B standard requires MC/DC (Modified Coverage/Decision Coverage)

Elements of a Strategy

- Common quality requirements that apply to all or most products
  - unambiguous definition and measures
- Set of documents normally produced during the quality process
  - contents and relationships
- Activities prescribed by the overall process
  - standard tools and practices
- Guidelines for project staffing and assignment of roles and responsibilities

Test and Analysis Plan

Answer the following questions:

1. What quality activities will be carried out?
2. What are the dependencies among the quality activities and between quality and other development activities?
3. What resources are needed and how will they be allocated?
4. How will both the process and the product be monitored?

Main Elements of a Plan

1. Items and features to be verified
   – Scope and target of the plan
2. Activities and resources
   – Constraints imposed by resources on activities
3. Approaches to be followed
   – Methods and tools
4. Criteria for evaluating results

Quality Goals

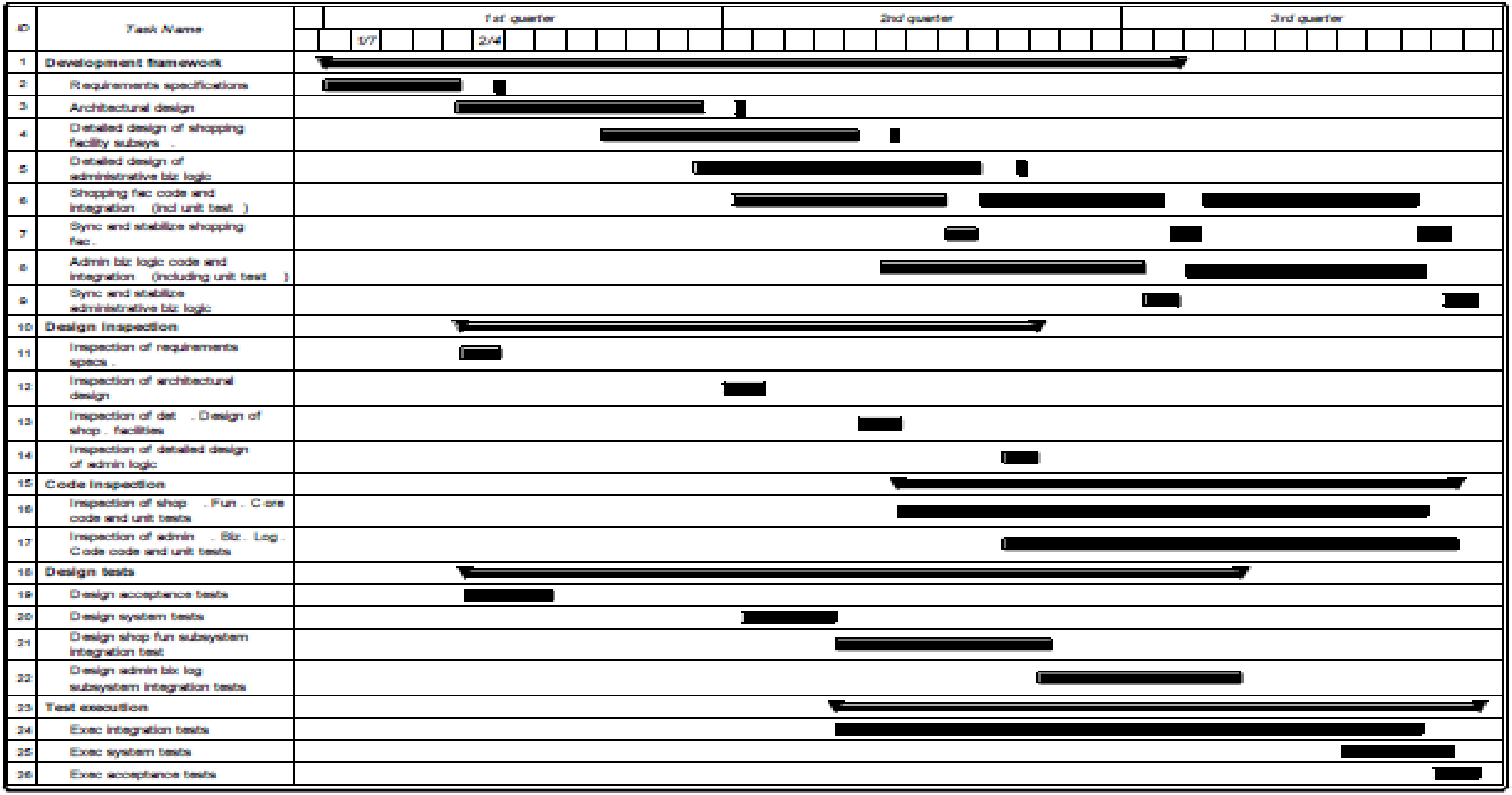
- Expressed as properties satisfied by the product
  - must include metrics to be monitored during the project
  - *example: before entering acceptance testing, the* product must pass comprehensive system testing with no critical or severe failures
  - not all details are available in the early stages of Development

- Initial plan
  - Based on incomplete information
  - Incrementally refined

Task Schedule

- Initially based on
  - quality strategy
  - past experience
- Breaks large tasks into subtasks
  - refine as process advances
- Includes dependencies
  - among quality activities
  - between quality and development activities
- Guidelines and objectives:
  - schedule activities for steady effort and continuous progress and evaluation without delaying development activities
  - Schedule activities as early as possible
  - Increase process visibility (how do we know we're on track?)

Sample Schedule



Schedule Risk

critical path = chain of activities that must be completed in sequence and that have maximum overall duration

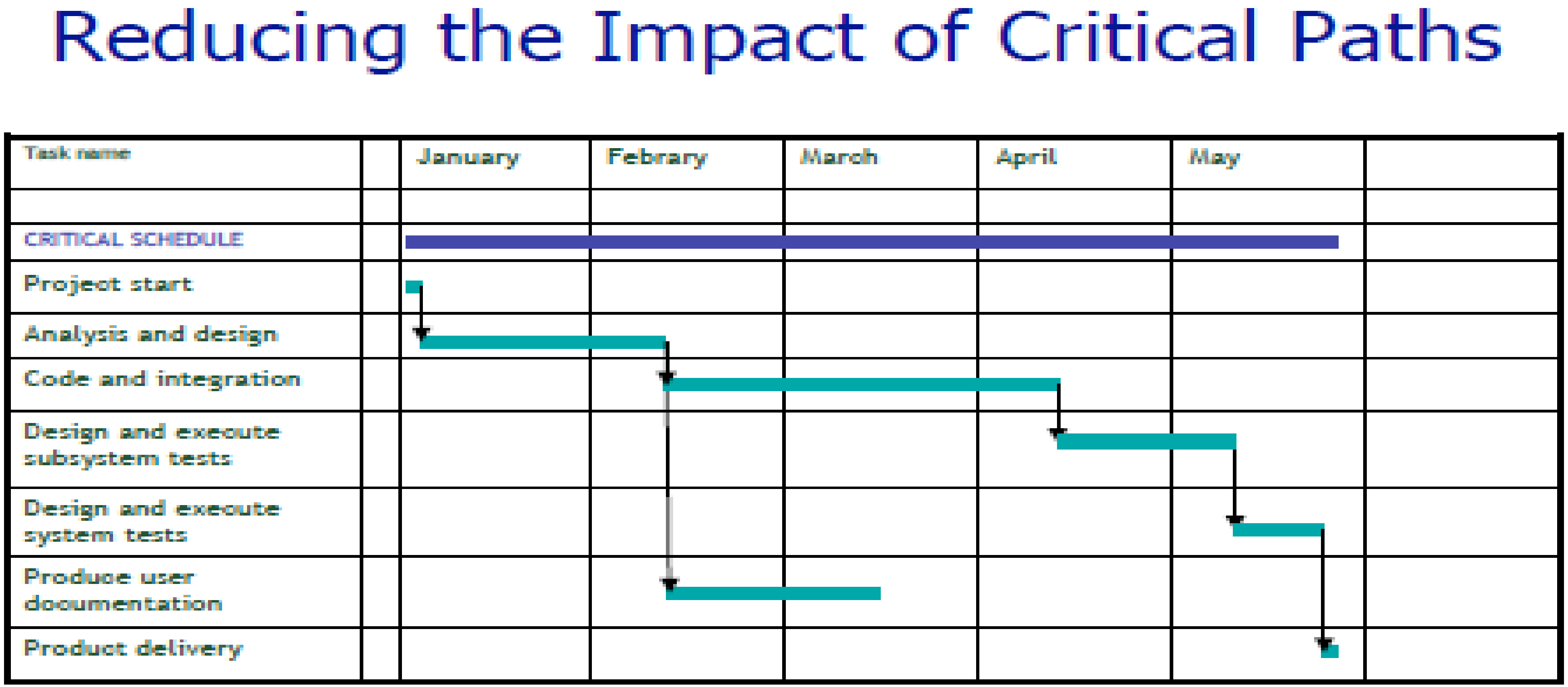
– Schedule critical tasks and tasks that depend on critical tasks as early as possible to

- provide schedule slack
- prevent delay in starting critical tasks
- *critical dependence = task on a critical path scheduled* immediately after some other task on the critical path

– May occur with tasks outside the quality plan (part of the project plan)

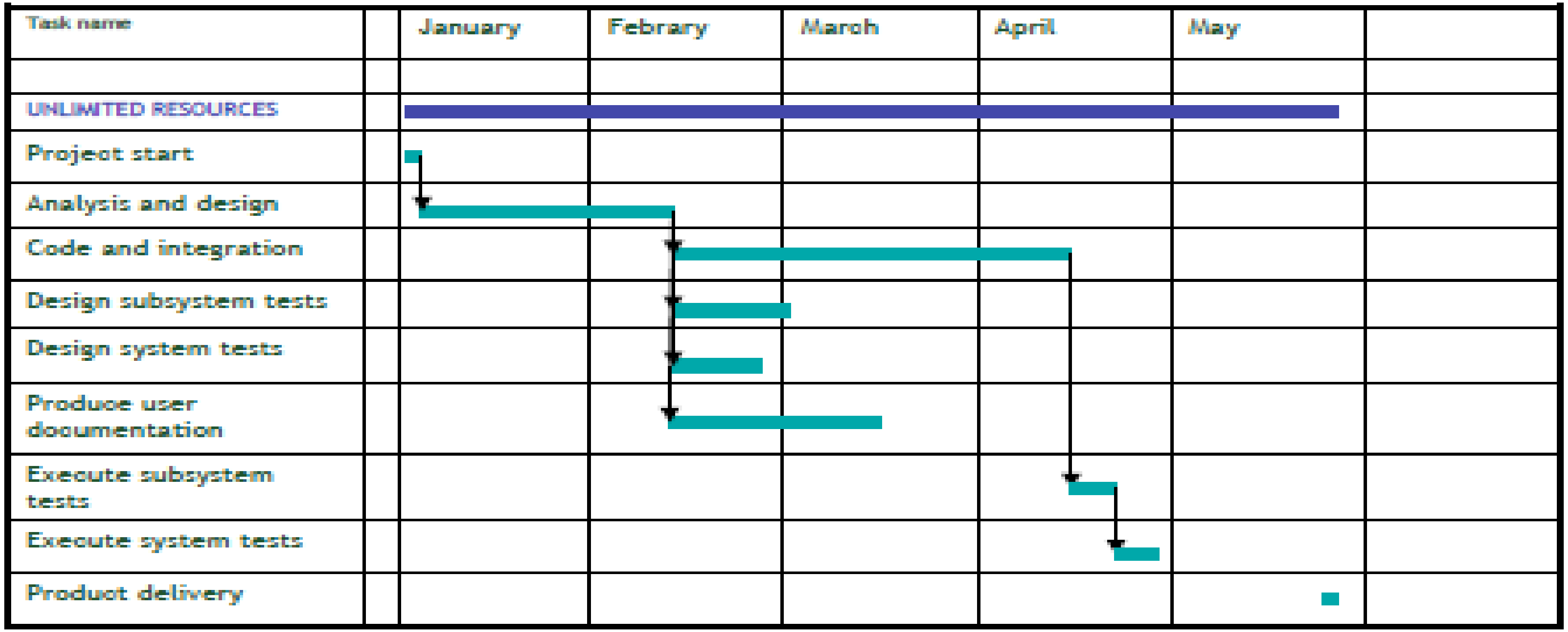
– Reduce critical dependences by decomposing tasks on critical path, factoring out subtasks that can be performed earlier

Reducing the Impact of Critical Paths



Cont...

Reducing the Impact of Critical Paths



Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk Planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Risk Planning – Risks generic to process Management

Risks cannot be eliminated, but they can be assessed, controlled, and monitored

- Generic management risk

– personnel

– technology

– schedule

- Quality risk

– development

– execution

– requirements

Personnel

Personnel

Example Risks

- Loss of a staff member
- Staff member under-qualified for task

Control Strategies

- cross training to avoid over-dependence on individuals
- continuous education
- identification of skills gaps early in project
- competitive compensation and promotion policies and rewarding work
- including training time in project schedule

Development

Development

Example Risks

- Poor quality software delivered to testing group
- Inadequate unit test and analysis before committing to the code base

Control Strategies

- Provide early warning and feedback
- Schedule inspection of design, code and test suites
- Connect development and inspection to the reward system
- Increase training through inspection
- Require coverage or other criteria at unit test level

Test Execution

Test Execution

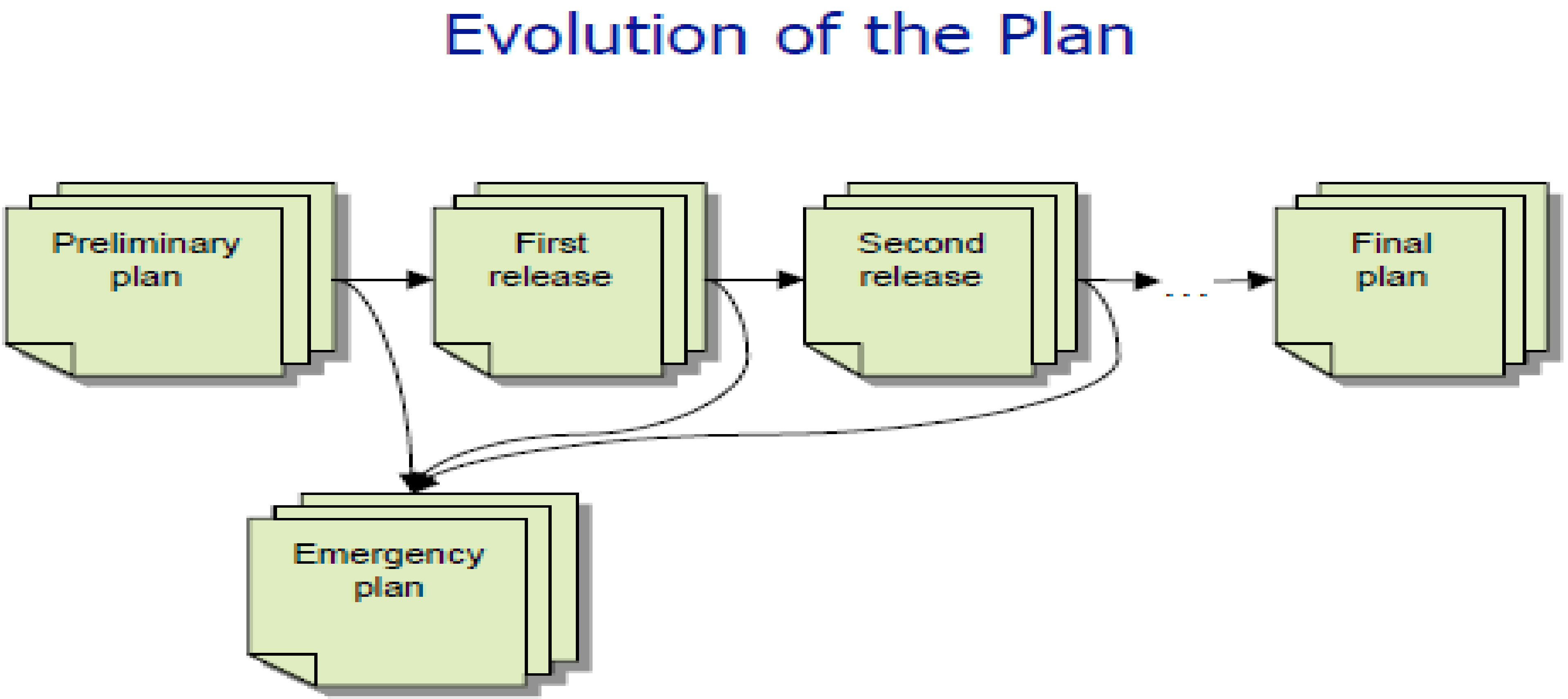
Example Risks

- Execution costs higher than planned
- Scarce resources available for testing

Control Strategies

- Minimize parts that require full system to be executed
- Inspect architecture to assess and improve testability
- Increase intermediate feedback
- Invest in scaffolding

Evolution of the Plan



Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

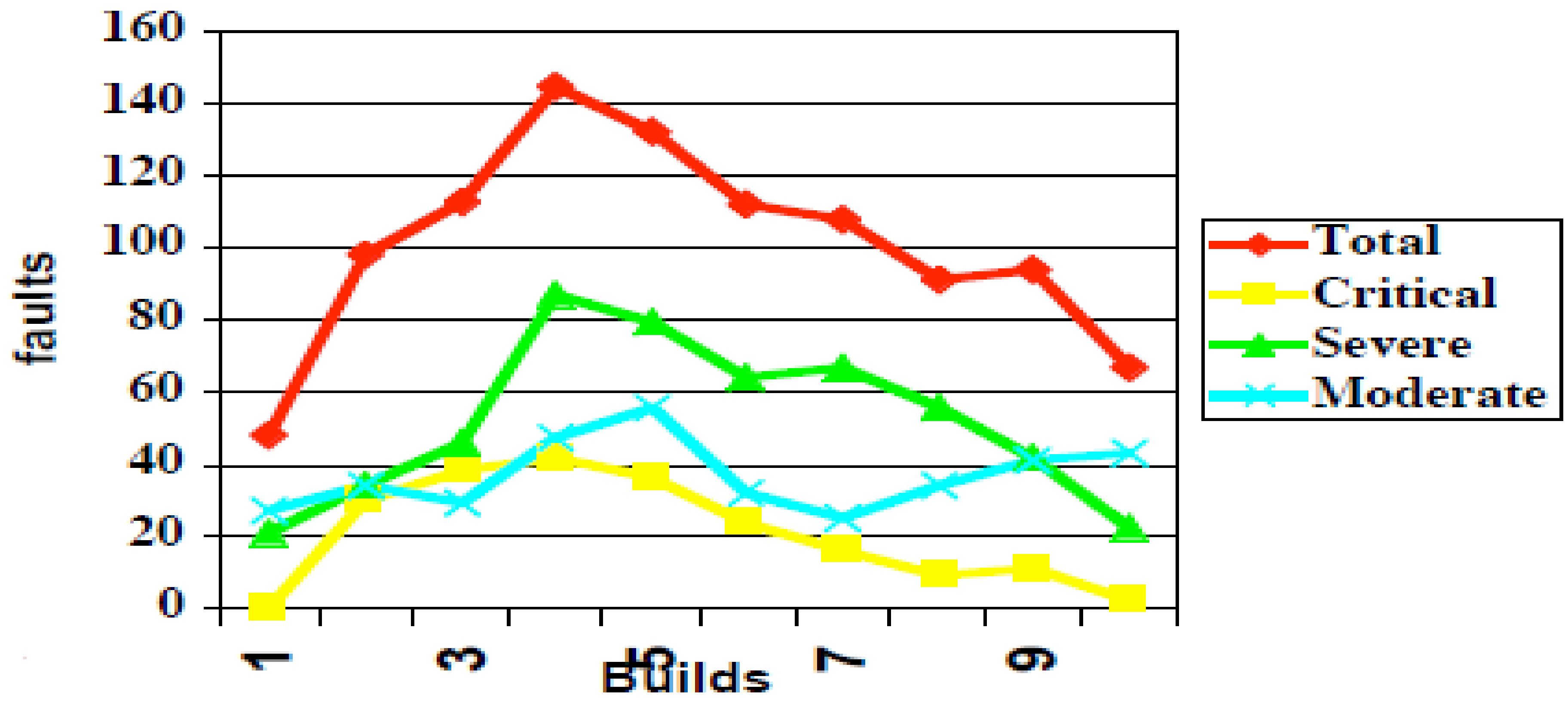
Process Monitoring

Process Monitoring

- Identify deviations from the quality plan as early as possible and take corrective action
- Depends on a plan that is
  - realistic
  - well organized
  - sufficiently detailed with clear, unambiguous milestones and criteria
- A process is *visible* to the extent that it can be effectively monitored

Typical Distribution of Faults for system builds through time

Evaluate Aggregated Data by Analogy



Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Process Improvement

Monitoring and improvement within a project or across multiple projects: Orthogonal Defect Classification (ODC) &Root Cause Analysis (RCA)

Orthogonal Defect Classification

Orthogonal Defect Classification (ODC)

- Accurate classification schema
  - for very large projects
  - to distill an unmanageable amount of detailed information
- Two main steps
  - Fault classification
    - when faults are detected
    - when faults are fixed
  - Fault analysis

ODC Fault Classification

ODC Fault Classification

When faults are detected

- *activity* executed when the fault is revealed
- *trigger* that exposed the fault
- *impact* of the fault on the customer

When faults are fixed

- *Target*: entity fixed to remove the fault
- *Type*: type of the fault
- *Source*: origin of the faulty modules (in-house, library, imported, outsourced)
- *Age* of the faulty element (new, old, rewritten, re-fixed code)

ODC activities and Triggers

ODC activities and triggers

- Review and Code Inspection
  - Design Conformance:
  - Logic/Flow
  - Backward Compatibility
  - Internal Document
  - Lateral Compatibility
  - Concurrency
  - Language Dependency
  - Side Effects
  - Rare Situation
- Structural (White Box) Test
  - Simple Path
  - Complex Path
- Functional (Black box) Test
  - Coverage
  - Variation
  - Sequencing
  - Interaction
- System Test
  - Workload/Stress
  - Recovery/Exception
  - Startup/Restart
  - Hardware Configuration
  - Software Configuration
  - Blocked Test

ODC Impact

ODC impact

- Installability
- Integrity/Security
- Performance
- Maintenance
- Serviceability
- Migration
- Documentation
- Usability
- Standards
- Reliability
- Accessibility
- Capability
- Requirements

ODC Fault Analysis

ODC Fault Analysis (example 1/4)

- Distribution of fault types versus activities
  - Different quality activities target different classes of faults
  - example:
    - algorithmic faults are targeted primarily by unit testing.
      - a high proportion of faults detected by unit testing should belong to this class
    - proportion of algorithmic faults found during unit testing
      - unusually small
      - larger than normal

      ⇒ unit tests may not have been well designed
    - proportion of algorithmic faults found during unit testing unusually large

    ⇒ integration testing may not focused strongly enough on interface faults

Cont...

ODC Fault Analysis (example 2/4)

- Distribution of triggers over time during field test
  - Faults corresponding to simple usage should arise early during field test, while faults corresponding to complex usage should arise late.
  - The rate of disclosure of new faults should asymptotically decrease
  - Unexpected distributions of triggers over time may indicate poor system or acceptance test
    - Triggers that correspond to simple usage reveal many faults late in acceptance testing

    $\Rightarrow$ The sample may not be representative of the user population
    - Continuously growing faults during acceptance test

    $\Rightarrow$ System testing may have failed

Cont...

ODC Fault Analysis (example 3/4)

- Age distribution over target code
  - *Most faults should be located in new and rewritten code*
  - The proportion of faults in new and rewritten code with respect to base and re-fixed code should gradually increase
  - Different patterns

  ⇒ may indicate holes in the fault tracking and removal process

  ⇒ may indicate inadequate test and analysis that failed in revealing faults early
  - Example
    - increase of faults located in base code after porting

    ⇒ may indicate inadequate tests for portability

Cont...

ODC Fault Analysis (example 4/4)

- Distribution of fault classes over time
  - The proportion of missing code faults should gradually decrease
  - The percentage of extraneous faults may slowly increase, because missing functionality should be revealed with use
    - increasing number of missing faults

    ⇒ may be a symptom of instability of the product
    - sudden sharp increase in extraneous faults

    ⇒ may indicate maintenance problems

Improving the Process

- Many classes of faults that occur frequently are rooted in process and development flaws
  - examples
    - Shallow architectural design that does not take into account resource allocation can lead to resource allocation faults
    - Lack of experience with the development environment, which leads to misunderstandings between analysts and programmers on rare and exceptional cases, can result in faults in exception handling.
- The occurrence of many such faults can be reduced by modifying the process and environment
  - examples
    - Resource allocation faults resulting from shallow architectural design can be reduced by introducing specific inspection tasks
    - Faults attributable to inexperience with the development environment can be reduced with focused training

Improving Current and Next Processes

- Identifying weak aspects of a process can be difficult
- Analysis of the fault history can help software engineers build a feedback mechanism to track relevant faults to their root causes
  - Sometimes information can be fed back directly into the current product development
  - More often it helps software engineers improve the development of future products

Root cause analysis (RCA)

- Technique for identifying and eliminating process faults
  - First developed in the nuclear power industry; used in many fields.
- Four main steps
  - *What* are the faults?
  - *When* did faults occur? When, and when were they found?
  - *Why* did faults occur?
  - *How* could faults be prevented?

What are the faults?

- Identify a class of important faults
- Faults are categorized by
  - severity = impact of the fault on the product
  - Kind
    - No fixed set of categories; Categories evolve and adapt
    - Goal:
      - Identify the few most important classes of faults and remove their causes
      - Differs from ODC: Not trying to compare trends for different classes of faults, but rather *focusing* on a few important classes

Fault Severity

Level	Description	Example
Critical	The product is unusable	The fault causes the program to crash
Severe	Some product features cannot be used, and there is no workaround	The fault inhibits importing files saved with a previous version of the program, and there is no workaround
Moderate	Some product features require workarounds to use, and reduce efficiency, reliability, or convenience and usability	The fault inhibits exporting in Postscript format. Postscript can be produced using the printing facility, but with loss of usability and efficiency
Cosmetic	Minor inconvenience	The fault limits the choice of colors for customizing the graphical interface, violating the specification but causing only minor inconvenience

Pareto Distribution (80/20)

- Pareto rule (80/20)
  - in many populations, a few (20%) are vital and many (80%) are trivial
- Fault analysis
  - 20% of the code is responsible for 80% of the faults
    - Faults tend to accumulate in a few modules
      - identifying potentially faulty modules can improve the cost effectiveness of fault detection
    - Some classes of faults predominate
      - removing the causes of a predominant class of faults can have a major impact on the quality of the process and of the resulting product

Why did faults occur?

- Core RCA step
  - trace representative faults back to causes
  - objective of identifying a "root" cause
- Iterative analysis
  - explain the error that led to the fault
  - explain the cause of that error
  - explain the cause of that cause
  - ...
- Rule of thumb
  - "ask why six times"

Example of fault tracing

- Tracing the causes of faults requires experience, judgment, and knowledge of the development process
- example
  - most significant class of faults = memory leaks
  - cause = forgetting to release memory in exception handlers
  - cause = lack of information: "Programmers can't easily determine what needs to be cleaned up in exception handlers"
  - cause = design error: "The resource management scheme assumes normal flow of control"
  - root problem = early design problem: "Exceptional conditions were an afterthought dealt with late in design"

How could faults be prevented?

- Many approaches depending on fault and process:
- From lightweight process changes
  - example
    - adding consideration of exceptional conditions to a design inspection checklist
- To heavyweight changes:
  - example
    - making explicit consideration of exceptional conditions a part of all requirements analysis and design steps

Goal is not perfection, but cost-effective improvement

Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

The Quality Team

- The quality plan must assign roles and responsibilities to people
- assignment of responsibility occurs at
  - strategic level
    - test and analysis strategy
    - structure of the organization
    - external requirements (e.g., certification agency)
  - tactical level
    - test and analysis plan

Roles and Responsibilities at Tactical Level

- balance level of effort across time
- manage personal interactions
- ensure sufficient accountability that quality tasks are not easily overlooked
- encourage objective judgment of quality
- prevent it from being subverted by schedule pressure
- foster shared commitment to quality among all team members
- develop and communicate shared knowledge and values regarding quality

Alternatives in Team Structure

- Conflicting pressures on choice of structure
  - example
    - autonomy to ensure objective assessment
    - cooperation to meet overall project objectives
- Different structures of roles and responsibilities
  - same individuals play roles of developer and tester
  - most testing responsibility assigned to a distinct group
  - some responsibility assigned to a distinct organization
- Distinguish
  - oversight and accountability for approving a task
  - responsibility for actually performing a task

Roles and responsibilities pros and cons

- Same individuals play roles of developer and tester
  - potential conflict between roles
    - example
      - a developer responsible for delivering a unit on schedule
      - responsible for integration testing that could reveal faults that delay delivery
  - requires countermeasures to control risks from conflict
- Roles assigned to different individuals
  - Potential conflict between individuals
    - example
      - developer and a tester who do not share motivation to deliver a quality product on schedule
  - requires countermeasures to control risks from conflict

Independent Testing Team

- Minimize risks of conflict between roles played by the same individual
  - Example
    - project manager with schedule pressures cannot
      - bypass quality activities or standards
      - reallocate people from testing to development
      - postpone quality activities until too late in the project
- Increases risk of conflict between goals of the independent quality team and the developers
- Plan
  - should include checks to ensure completion of quality activities
  - Example
    - developers perform module testing
    - independent quality team performs integration and system testing
    - quality team should check completeness of module tests

Managing Communication

- Testing and development teams must share the goal of shipping a high-quality product on schedule
  - testing team
    - must not be perceived as relieving developers from responsibility for quality
    - should not be completely oblivious to schedule pressure
- Independent quality teams require a mature development process
  - Test designers must
    - work on sufficiently precise specifications
    - execute tests in a controllable test environment
- Versions and configurations must be well defined
- Failures and faults must be suitably tracked and monitored across versions

Testing within XP

- Full integration of quality activities with development
  - Minimize communication and coordination overhead
  - Developers take full responsibility for the quality of their work
  - Technology and application expertise for quality tasks match expertise available for development tasks
- Plan
  - check that quality activities and objective assessment are not easily tossed aside as deadlines loom
  - example
    - XP "test first" together with pair programming guard against some of the inherent risks of mixing roles

Outsourcing Test and Analysis

- (Wrong) motivation
  - testing is less technically demanding than development and can be carried out by lower-paid and lower-skilled individuals
- Why wrong
  - confuses test execution (straightforward) with analysis and test design (as demanding as design and programming)
- A better motivation
  - to maximize independence
    - and possibly reduce cost as (only) a secondary effect
- The plan must define
  - milestones and delivery for outsourced activities
  - checks on the quality of delivery in both directions

Summary

- Planning is necessary to
  - order, provision, and coordinate quality activities
    - coordinate quality process with overall development
    - includes allocation of roles and responsibilities
  - provide unambiguous milestones for judging progress
- Process *visibility* is key
  - ability to monitor quality and schedule at each step
    - intermediate verification steps: because cost grows with time between error and repair
  - monitor risks explicitly, with contingency plan ready
- *Monitoring* feeds process improvement
  - of a single project, and across projects

Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Documenting Analysis and Test

Why Produce Quality Documentation?

- Monitor and assess the process
  - For internal use (*process visibility*)
  - For external authorities (certification, auditing)
- Improve the process
  - Maintain a body of knowledge reused across projects
  - Summarize and present data for process improvement
- Increase reusability of test suites and other artifacts within and across projects

Major categories of documents

- Planning documents
  - describe the organization of the quality process
  - include organization *strategies* and project *plans*
- Specification documents
  - describe test suites and test cases
    (as well as artifacts for other quality tasks)
  - test design specifications, test case specification, checklists, analysis procedure specifications
- Reporting documents
  - Details and summary of analysis and test results

Metadata

- Documents should include *metadata* to facilitate management
  - **Approval:** persons responsible for the document
  - **History** of the document
  - **Table of Contents**
  - **Summary:** relevance and possible uses of the document
  - **Goals:** purpose of the document- Who should read it, and why?
  - **Required documents and references:** reference to documents and artifacts needed for understanding and exploiting this document
  - **Glossary:** technical terms used in the document

Metadata example: Chipmunk Document Template

Document Title

Approvals

issued by	*name* *signature* *date*
approved by	*name* *signature* *date*
distribution status	*(internal use only, restricted, ...)*
distribution list	*(people to whom the document must be sent)*

History

version	description

Metadata may be provided or managed by tools. For example, version control system may maintain version history.

Chipmunk Document Template (continued)

....

Table of Contents

List of sections

Summary

Summarize the contents of the document. The summary should clearly explain the relevance of the document to its possible uses.

Goals of the document

Describe the purpose of this document: Who should read it, and why?

Required documents and references

Provide a reference to other documents and artifacts needed for understanding and exploiting this document. Provide a rationale for the provided references.

Glossary

Provide a glossary of terms required to understand this document.

Section 1

. . .

Section N

. . .

Naming conventions

- Naming conventions help people identify documents quickly
- A typical standard for document names include keywords indicating
  - general scope of the document (project and part)
  - kind of document (for example, test plan)
  - specific document identity
  - version

Sample naming standard

Project or product (e.g., W for "web presence")

Sub-project (e.g., "Business logic")

Item type

Identifier

Version

W B XX – YY.ZZ

example:

W B 12 – 22 .04

Might specify version 4 of document 12-22 (quality monitoring procedures for third-party software components) of web presence project, business logic subsystem.

Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Analysis and test strategy

- Strategy document describes quality guidelines for sets of projects
  (usually for an entire company or organization)
- Varies among organizations
- Few key elements:
  common quality requirements across products
- May depend on business conditions - examples
  - safety-critical software producer may need to satisfy minimum dependability requirements defined by a certification authority
  - embedded software department may need to ensure portability across product lines
- Sets out *requirements on other quality documents*

Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Analysis and Test Plan

- Standardized structure see next slide
- Overall quality plan comprises several individual plans
  - Each individual plan indicates the items to be verified through analysis or testing
  - Example: documents to be inspected, code to be analyzed or tested, ...
- May refer to the whole system or part of it
  - Example: subsystem or a set of units
- May not address all aspects of quality activities
  - Should indicate features to be verified and excluded
    - Example: for a GUI– might deal only with functional properties and not with usability (if a distinct team handles usability testing)
  - Indication of excluded features is important
    - omitted testing is a major cause of failure in large projects

Standard Organization of a Plan

- Analysis and test items: items to be tested or analyzed
- Features to be tested: features considered in the plan
- Features not to be tested: Features not considered in the plan
- Approach: overall analysis and test approach
- Pass/Fail criteria: Rules that determine the status of an artifact
- Suspension and resumption criteria: Conditions to trigger suspension of test and analysis activities
- Risks and contingencies: Risks foreseen and contingency plans
- Deliverables: artifacts and documents that must be produced
- Task and schedule: description of analysis and test tasks (usually includes GANTT and PERT diagrams)
- Staff and responsibilities
- Environmental needs: Hardware and software

Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Test Design Specification Documents

- Same purpose as other software design documentation:
  - Guiding further development
  - Preparing for maintenance
- Test design specification documents:
  - describe complete test suites
  - may be divided into
    - unit, integration, system, acceptance suites (organize by granularity)
    - functional, structural, performance suites (organized by objectives)
    - ...
  - include all the information needed for
    - initial selection of test cases
    - maintenance of the test suite over time
  - identify features to be verified (cross-reference to specification or design document
  - include description of testing procedure and pass/fail criteria (references to scaffolding and oracles)
  - includes (logically) a list of test cases

Test case specification document

- Complete test design for individual test case
- Defines
  - test inputs
  - required environmental conditions
  - procedures for test execution
  - expected outputs
- Indicates
  - item to be tested (reference to design document)
- Describes dependence on execution of other test cases
- Is labeled with a unique identifier

Agenda

- ✓Planning and Monitoring
- ✓Quality and process
- ✓Test and analysis strategies and plans
- ✓Risk planning
- ✓Monitoring the process
- ✓Improving the process
- ✓The quality team
- ✓Organizing documents
- ✓Test strategy document
- ✓Analysis and test plan
- ✓Test design specifications documents
- ✓Test and analysis reports
- ✓Conclusion

Test and Analysis Reports

- Report test and analysis results
- Serve
  - Developers
    - identify open faults
    - schedule fixes and revisions
  - Test designers
    - assess and refine their approach see chapter 20
- Prioritized list of open faults: the core of the fault handling and repair procedure
- Failure reports must be
  - consolidated and categorized to manage repair effort systematically
  - prioritized to properly allocate effort and handle all faults

Summary reports and detailed logs

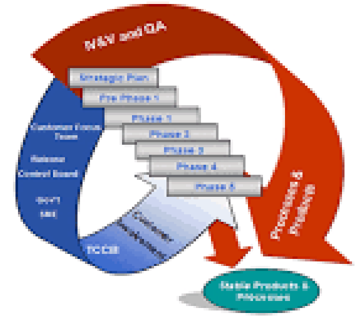
- Summary reports track progress and status
  - may be simple confirmation that build-and-test cycle ran successfully
  - may provide information to guide attention to trouble spots
- Include summary tables with
  - executed test suites
  - number of failures
  - breakdown of failures into
    - repeated from prior test execution,
    - new failures
    - test cases that previously failed but now execute correctly
- May be prescribed by a certifying authority

Conclusion

In a nut shell we have seen a Planning and Monitoring, Quality and process, Test and analysis strategies and plans, Risk planning, Monitoring the process, Improving the process, The quality team, Organizing documents, Test strategy document, Analysis and test plan, Test design specifications documents and Test and analysis reports

Software Testing

Automated Testing
Manual Testing
Software Quality Assurance



Module - 5: Integration and Component-Based Software Testing

By

Dr.Manjunath T N
Professor

Agenda

1. Integration Testing Strategies
2. Testing Components and assemblies
3. System Testing
4. Acceptance Testing
5. Regression Testing
6. Usability Testing
7. Regression Testing Selection Techniques
8. Test Case prioritization and Selective Execution
9. Levels of Testing and Integration Testing
10. Traditional view of testing levels
11. Alternative life cycle models
12. The SATM System
13. Separating Integration and System Testing
14. A Closer look at the SATM System
15. Decomposition Based
16. Call Graph Based
17. Path Based Integrations

Integration Testing Strategies

- Bottom - up testing (test harness).
- Top - down testing (stubs).
- Modified top - down testing - test levels independently.
- Big Bang.
- Sandwich testing.

Top-Down Integration Testing

- Main program used as a test driver and stubs are substitutes for components directly subordinate to it.
- Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
- Tests are conducted as each component is integrated.
- On completion of each set of tests and other stub is replaced with a real component.
- Regression testing may be used to ensure that new errors not introduced.

Bottom-Up Integration Testing

- Low level components are combined in clusters that perform a specific software function.
- A driver (control program) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.

	Bottom - Up	Top - Down	Big Bang	Sandwich
Integration	Early	Early		Early
Time to get working program	Late	Early	Late	Early
Drivers	Yes	No	Yes	Yes
Stub	No	Yes	Yes	Yes
Parallelism	Medium	Low	High	Medium
Test specification	Easy	Hard	Easy	Medium
Product control seq.	Easy	Hard	Easy	Hard

Testing Components and assemblies

Working Definition of *Component*

- <u>Reusable</u> unit of <u>deployment</u> and <u>composition</u>
  - Deployed and integrated multiple times
  - Integrated by different teams (usually)
    - Component producer is distinct from component user
- Characterized by an *interface* or *contract*
    - Describes access points, parameters, and all functional and non-functional behavior and conditions for using the component
    - No other access (e.g., source code) is usually available
- Often larger grain than objects or packages
  - Example: A complete database system may be a component

Components — Related Concepts

- Framework
  - Skeleton or micro-architecture of an application
  - May be packaged and reused as a component, with “hooks” or “slots” in the interface contract
- Design patterns
  - Logical design fragments
  - Frameworks often implement patterns, but patterns are not frameworks. Frameworks are concrete, patterns are abstract
- Component-based system
  - A system composed primarily by assembling components, often “Commercial off-the-shelf” (COTS) components
  - Usually includes application-specific “glue code”

Component Interface Contracts

- Application programming interface (API) is distinct from implementation
  - Example: DOM interface for XML is distinct from many possible implementations, from different sources
- Interface includes *everything* that must be known to use the component
  - More than just method signatures, exceptions, etc
  - May include non-functional characteristics like performance, capacity, security
  - May include dependence on other components

Challenges in Testing Components

- The component builder's challenge:
  - Impossible to know all the ways a component may be used
  - Difficult to recognize and specify all potentially important properties and dependencies

- The component user's challenge:
  - No visibility "inside" the component
  - Often difficult to judge suitability for a particular use and context

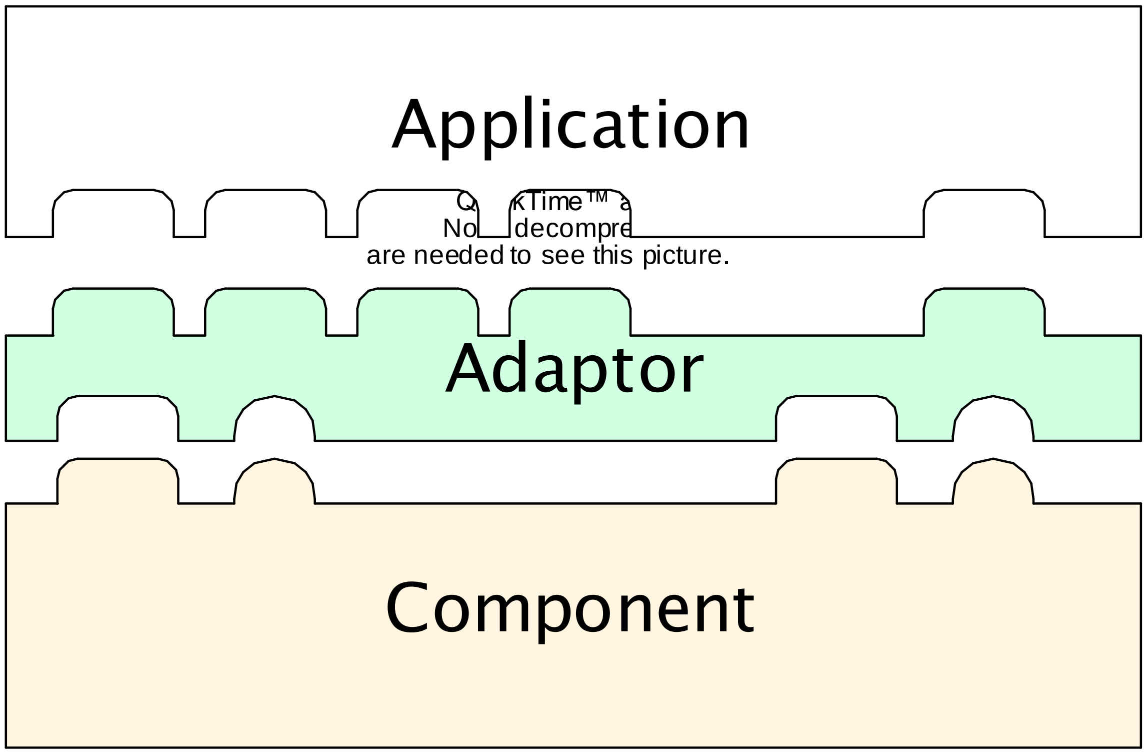
Testing a Component: Producer View

- First: Thorough unit and subsystem testing
  - Includes thorough functional testing based on application program interface (API)
  - Rule of thumb: Reusable component requires at least twice the effort in design, implementation, and testing as a subsystem constructed for a single use (often more)
- Second: Thorough acceptance testing
  - Based on scenarios of expected use
  - Includes stress and capacity testing
    - Find and document the limits of applicability

Testing a Component: User View

- Not primarily to find faults in the component
- Major question: Is the component suitable for *this* application?
  - Primary risk is not fitting the application context:
    - Unanticipated dependence or interactions with environment
    - Performance or capacity limits
    - Missing functionality, misunderstood API
  - Risk high when using component for first time
- Reducing risk: Trial integration early
  - Often worthwhile to build driver to test model scenarios, long before actual integration

Adapting and Testing a Component



- Applications often access components through an adaptor, which can also be used by a test driver

System Testing

- Recovery testing
  - checks system's ability to recover from failures
- Security testing
  - verifies that system protection mechanism prevents improper penetration or data alteration
- Stress testing
  - program is checked to see how well it deals with abnormal resource demands
- Performance testing
  - tests the run-time performance of software

Acceptance Testing

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test
  - version of the complete software is tested by customer under the supervision of the developer at the developer's site
- Beta test
  - version of the complete software is tested by customer at his or her own site without the developer being present

Acceptance Testing Approaches

- Benchmark test.
- Pilot testing.
- Parallel testing.

Regression Testing

- Check for defects propagated to other modules by changes made to existing program
  - Representative sample of existing test cases is used to exercise all software functions.
  - Additional test cases focusing software functions likely to be affected by the change.
  - Tests cases that focus on the changed software components.

Usability Testing

Usability testing is a technique used in user-centered interaction design to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system. This is in contrast with usability inspection methods where experts use different methods to evaluate a user interface without involving users.

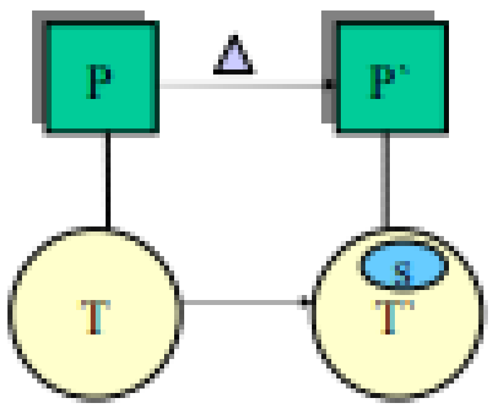
Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose. Examples of products that commonly benefit from usability testing are food, consumer products, web sites or web applications, computer interfaces, documents, and devices.

Usability testing measures the usability, or ease of use, of a specific object or set of objects, whereas general human–computer interaction studies attempt to formulate universal principles.

Regression Testing Selection Techniques

Regression testing is a necessary and expensive maintenance task performed on modified programs to ensure that the changes have not adversely effected the unchanged code of the program.
One strategy is to rerun the entire test suit on the changed program.
This is a heavy resource and time consuming process.
A solution to this is: Regression test selection techniques: selects a subset of test cases, thus reducing the time and resources required.

Selection Techniques:



Most of the selection techniques are based on the information about the code of the program and the modified version. Some however are based on the program specifications.

Following are some of the code based techniques, which are used for this study

Selection technique algorithms used for study

Safe: selects all the test cases that cover/execute the changed methods at least once.
Minimization: selects a minimum set of test cases that execute all the changed methods.
Random25: selects randomly 25% of the total test cases. Random50: selects randomly 50% of the total test cases.
Random75: selects randomly 75% of the total test cases.

Test Case prioritization and Selective Execution

Regression testing activities such as test case selection and test case prioritization are ordinarily based on the criteria which focused around code coverage, code modifications and test execution costs. The approach mainly based on the multiple criteria of code coverage which performs efficient selection of test case. The method mainly aims to maximize the coverage size by executing the test cases effectively

The goal of regression testing is to ensure that changes to the system have not introduced errors. One approach is to rerun all the test cases in the existing test suite and check for new faults. But rerunning the entire test suite is often too costly.

To make the execution of test cases more cost effective, two major approaches are made use of. They are the Regression Test Selection (RTS) and Regression Test Prioritization (RTP) techniques.

Many RTS and RTP techniques consider a single criterion for optimization of test cases. But, the use of a single criterion severely limits the ability of the resulting regression test suite to locate faults. Harman et al., induce the need of multiple criteria and provides a list of criteria with different weights.

The two criteria for selection are code coverage and sum coverage of the program. Code coverage assumes that there exist test cases that effectively cover the changed area of code of the software. Sum coverage is a new approach that maximizes the minimum sum of coverage across all software elements.

The selected test cases are prioritized using a greedy algorithm to maximize the minimum sum of coverage across all software elements.

Traditional View of Testing Levels

The traditional model of software development is the Waterfall model, which is drawn as a V in. In this view, information produced in one of the development phases constitutes the basis for test case identification at that level.

Nothing controversial here: we certainly would hope that system test cases are somehow correlated with the requirements specification, and that unit test cases are derived from the detailed design of the unit. Two observations: there is a clear presumption of functional testing here, and there is an implied “bottom-up” testing order.

Alternative Life Cycle Models

Since the early 1980s, practitioners have devised alternatives in response to shortcomings of the traditional waterfall model of software development Common to all of these alternatives is the shift away from the functional decomposition to an emphasis on composition. Decomposition is a perfect fit both to the top-down progression of the waterfall model and to the bottom-up testing order.

One of the major weaknesses of waterfall development cited by is the over-reliance on this whole paradigm. Functional decomposition can only be well done when the system is completely understood, and it promotes analysis to the near exclusion of synthesis. The result is a very long separation between requirements specification and a completed system, and during this interval, there is no opportunity for feedback from the customer. Composition, on the other hand, is closer the way people work: start with something known and understood, then add to it gradually, and maybe remove undesired portions.

There is a very nice analogy with positive and negative sculpture. In negative sculpture, work proceeds by removing unwanted material, as in the mathematician's view of sculpting Michelangelo's David: start with a piece of marble, and simply chip away all non-David. Positive sculpture is often done with a medium like wax.

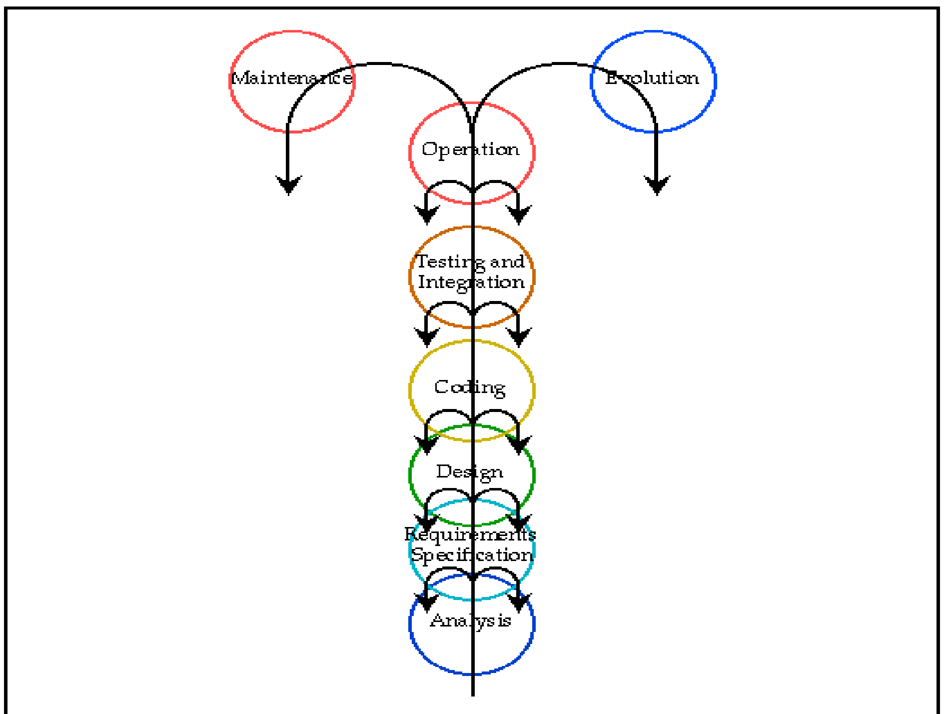
The central shape is approximated, and then wax is either added or removed until the desired shape is attained. Think about the consequences of a mistake: with negative sculpture, the whole work must be thrown away, and restarted. With positive sculpture, the erroneous part is simply removed and replaced. The centrality of composition in the alternative models has a major implication for integration testing.

Waterfall Spin-offs

There are three mainline derivatives of the waterfall model: incremental development, evolutionary development, and the Spiral model [Boehm 88]. Each of these involves a series of increments or builds, Within a build, the normal waterfall phases from detailed design through testing occur, with one important difference: system testing is split into two steps, regression and progression testing

An Object-Oriented Life Cycle Model

When software is developed with an object orientation, none of our life cycle models fit very well. The main reasons: the object orientation is highly compositional in nature, and there is dense interaction among the construction phases of object-oriented analysis, object-oriented design, and object-oriented programming. We could show this with pronounced feedback loops among waterfall phases, but the fountain model [Henderson-Sellers 90] is a much more appropriate metaphor. In the fountain model, the foundation is the requirements analysis of real world systems



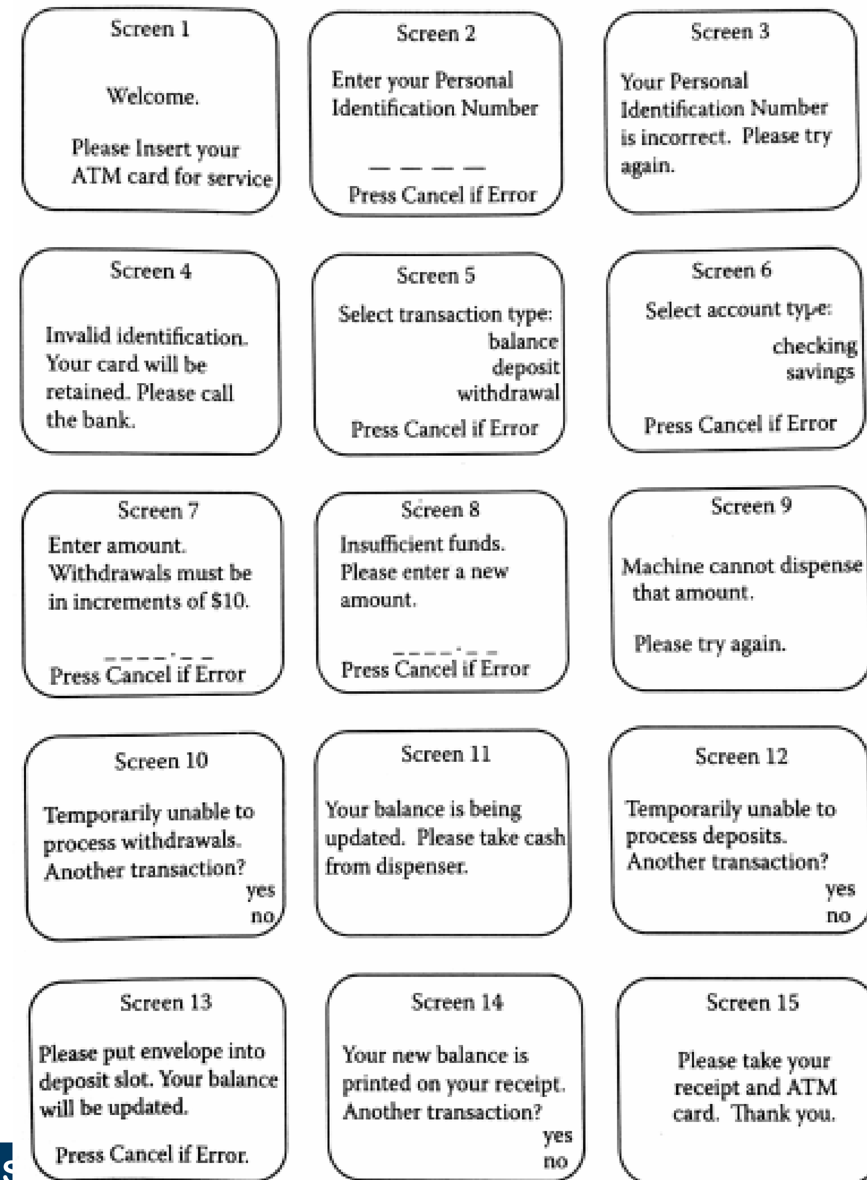
Formulations of the SATM System

The Simple Automatic Teller Machine (SATM) system. there are function buttons B1, B2, and B3, a digit keypad with a cancel key, slots for printer receipts and ATM cards, and doors for deposits and cash withdrawals. The SATM system is described here in two ways: with a structured analysis approach, and with an object-oriented approach. These descriptions are not complete, but they contain detail sufficient to illustrate the testing techniques under discussion.

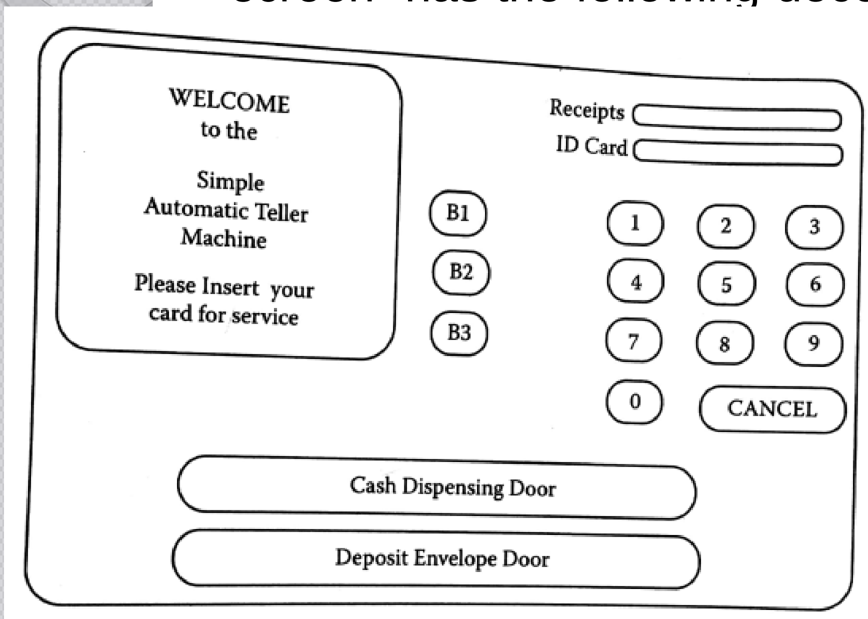
SATM with Structured Analysis

The structured analysis approach to requirements specification is the most widely used method in the world. It enjoys extensive CASE tool support as well as commercial training, and is described in numerous texts. The technique is based on three complementary models: function, data, and control. Here we use data flow diagrams for the functional models, entity/relationship models for data, and finite state machine models for the control aspect of the SATM system. The functional and data models were drawn with the Deft CASE tool from Sybase Inc. That tool identifies external devices (such as the terminal doors) with lower case letters, and elements of the functional decomposition with numbers (such as 1.5 for the Validate Card function).

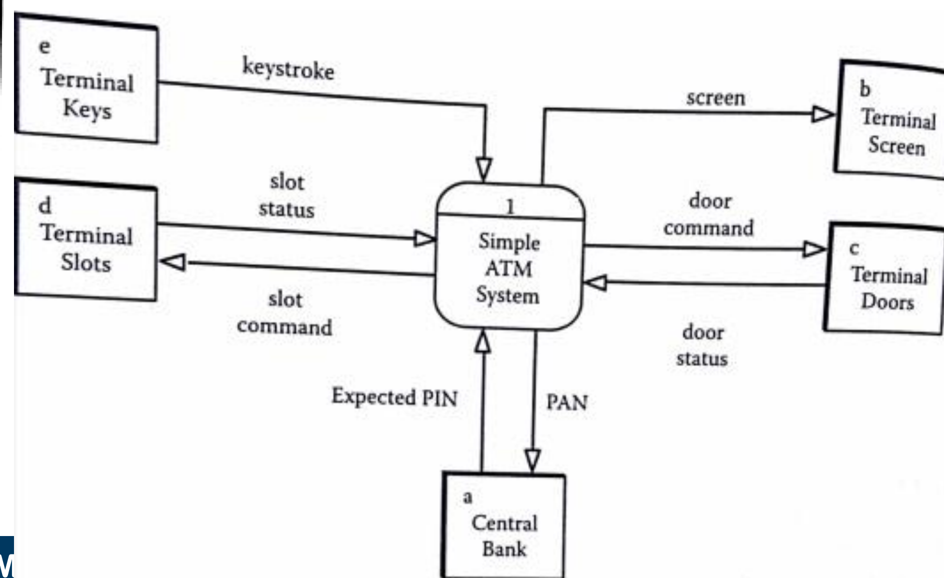
The open and filled arrowheads on flow arrows signify whether the flow item is simple or compound. The portions of the SATM system shown here pertain generally to the personal identification number (PIN) verification portion of the system.



The Deft CASE tool distinguishes between simple and compound flows, where compound flows may be decomposed into other flows, which may themselves be compound. The graphic appearance of this choice is that simple flows have filled arrowheads, while compound flows have open arrowheads. As an example, the compound flow "screen" has the following decomposition



The SATM Terminal



Context Diagram of the SATM System

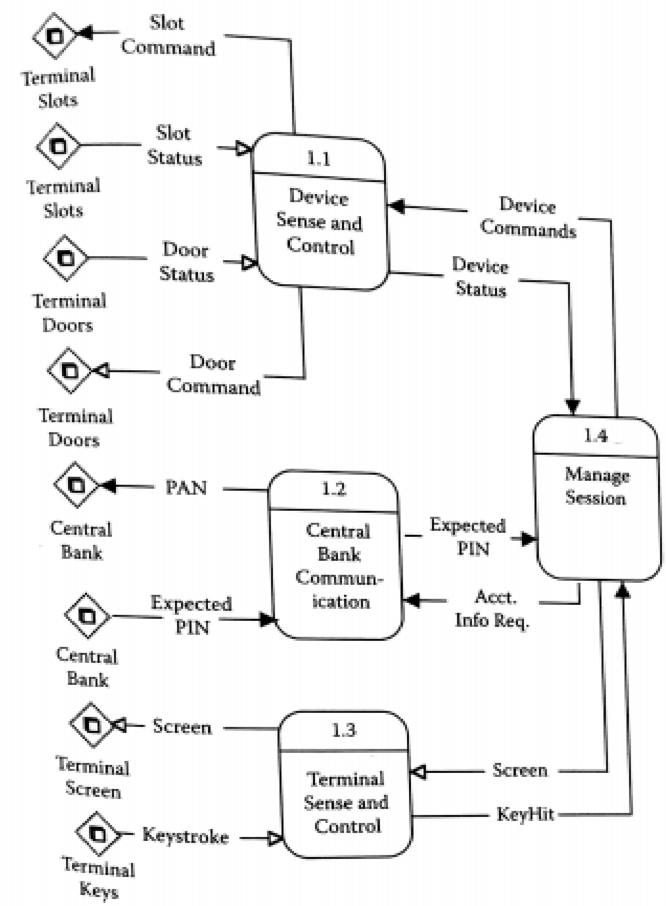
The Structured Analysis approach models shown here are not complete but they contain sufficient details to illustrate the testing techniques.

The Structured analysis approach to requirements specifications is still widely used.

It Enjoys extensive CASE tool support.

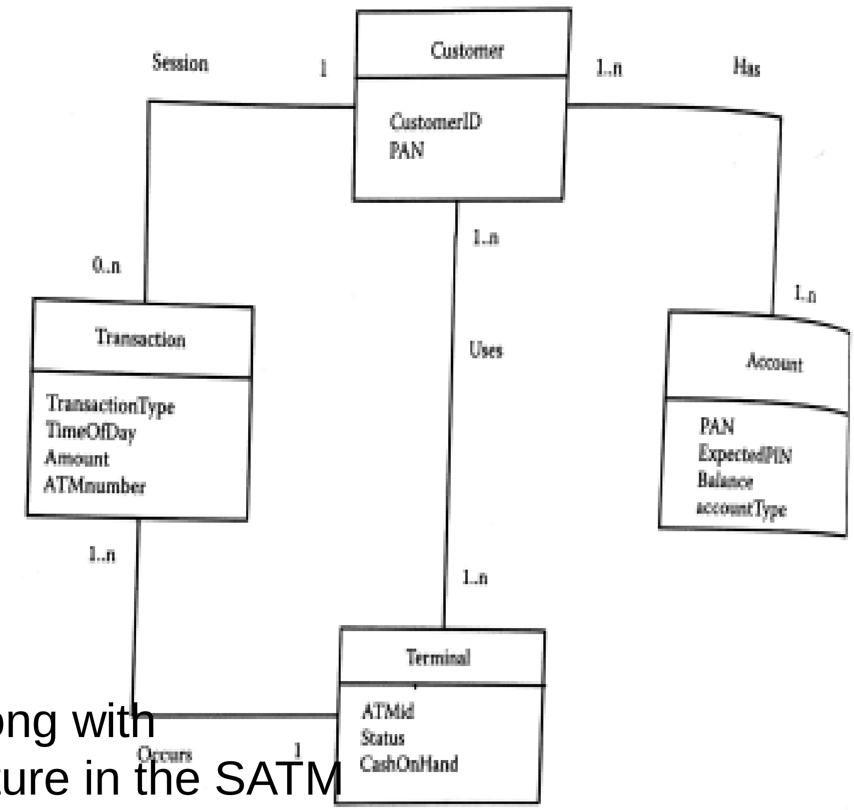
The Techniques used are based on three complementary models: function, data and control.

Here we use dataflow diagrams for functional model, the entity relationship model for data and finite state machine models for the control aspects of SATM



Level-1 Dataflow Diagram of the SATM System

screen3	wrong PIN
screen4	PIN failed, card retained
screen5	select trans type
screen6	select account type
screen7	enter amount
screen8	insufficient funds
screen9	cannot dispense that amount
screen10	cannot process withdrawals
screen11	take your cash
screen12	cannot process deposits
screen13	put dep envelop in slot
screen14	another transaction?
screen15	Thanks; take card and receipt



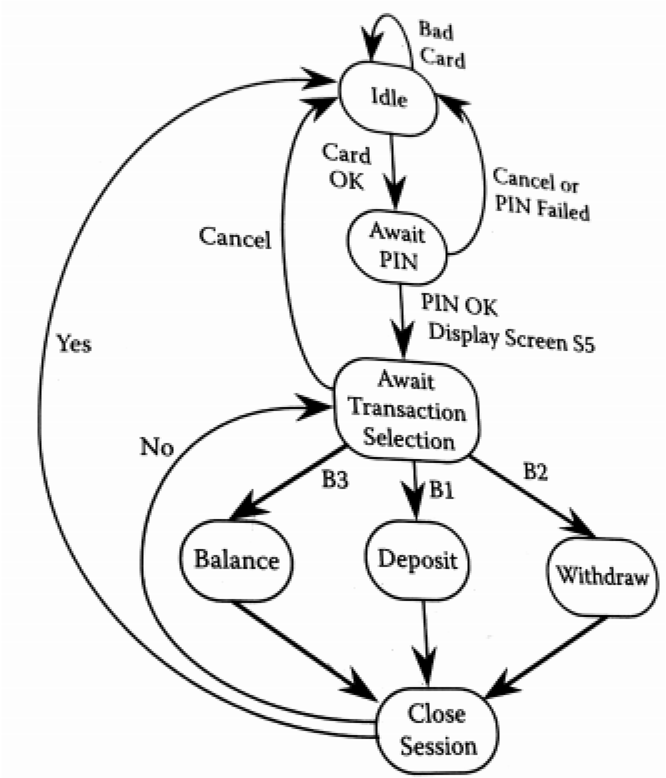
The Different Screens are shown along with E-R diagram of the major data structure in the SATM

- -Customer
- -Accounts
- -Terminals
- -Transactions.

E-R Model of the SATM System

The Upper level finite state machine which divides the system into states that correspond to stages of customer usage.
Other Choices are possible for instance, we might choose states to be screens displayed.

Finite state machines can be hierarchically decomposed in much the same way as dataflow diagrams can.



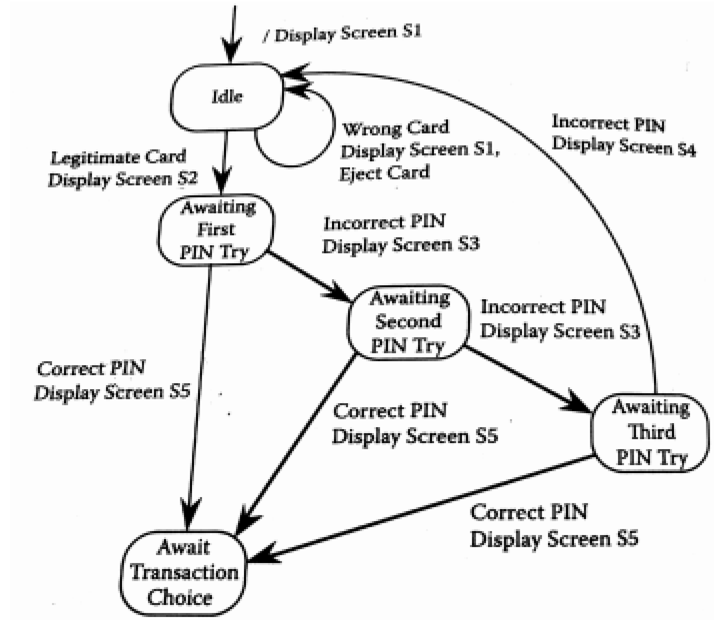
Upper Level SATM Finite State Machine

The Decomposition of the Await PIN state. Here the state transitions are caused either by events at the ATM terminal or by data conditions.

When a transition occurs a corresponding action may also occur.

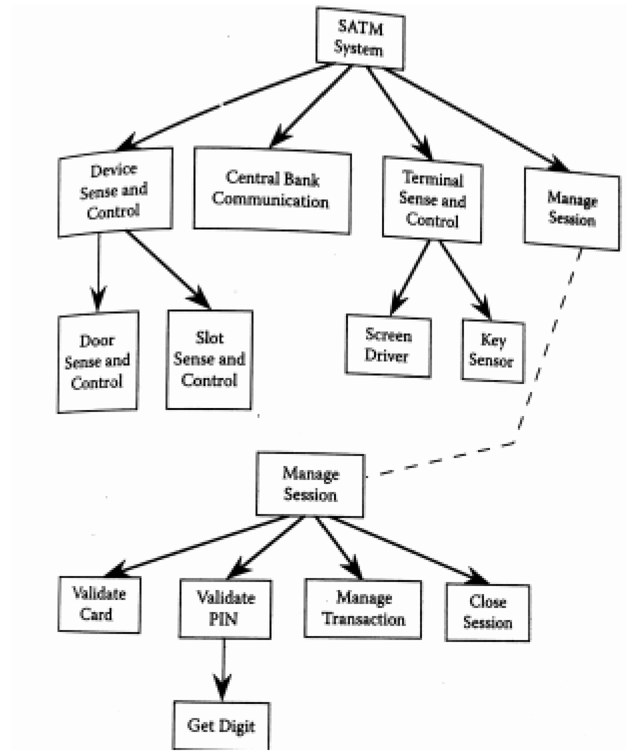
We choose to use screen displays as such actions, this choice will prove to be very handy when we develop system-level test cases.

The function, data and control models are the basis for design activities in the waterfall model



PIN Entry finite State Machine

```
1       SATM System
1.1     Device Sense & Control
1.1.1 Door Sense & Control
1.1.1.1       Get Door Status
1.1.1.2       Control Door
1.1.1.3       Dispense Cash
1.1.2 Slot Sense & Control
1.1.2.1       WatchCardSlot
1.1.2.2       Get Deposit Slot Status
1.1.2.3       Control Card Roller
1.1.2.3       Control Envelope Roller
1.1.2.5       Read Card Strip
1.2     Central Bank Comm.
1.2.1 Get PIN for PAN
1.2.2 Get Account Status
1.2.3 Post Daily Transactions
1.3     Terminal Sense & Control
1.3.1 Screen Driver
1.3.2 Key Sensor
1.4     Manage Session
1.4.1 Validate Card
1.4.2 Validate PIN
1.4.2.1       GetPIN
1.4.3 Close Session
1.4.3.1       New Transaction Request
1.4.3.2       Print Receipt
1.4.3.3       Post Transaction Local
1.4.4 Manage Transaction
1.4.4.1       Get Transaction Type
1.4.4.2       Get Account Type
1.4.4.3       Report Balance
1.4.4.4       Process Deposit
1.4.4.5       Process Withdrawal
```

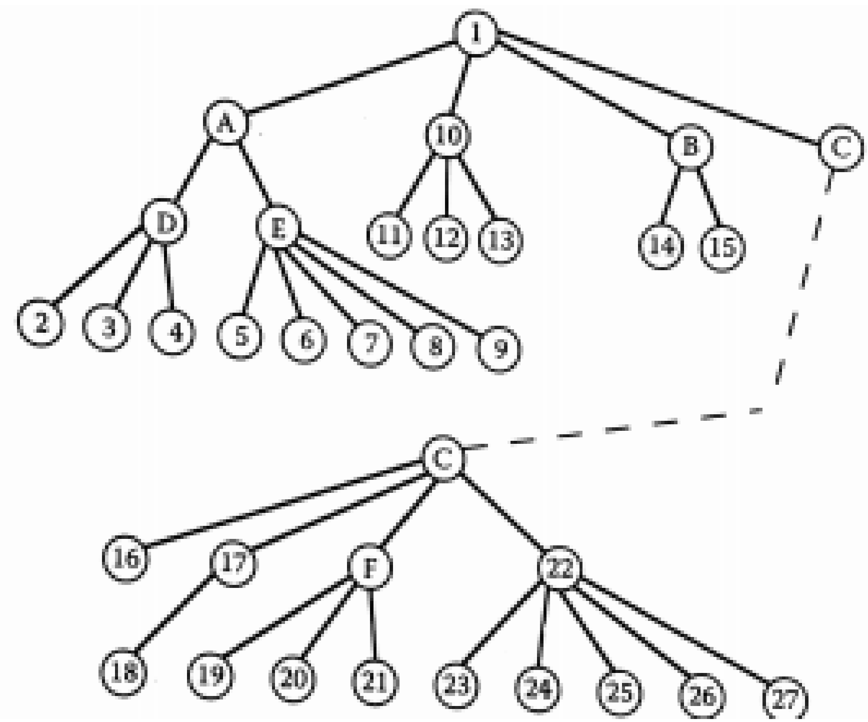


A Decomposition tree for the SATM System

The Pseudocode shown here is for SATM system and it is decomposed into tree structure for different functionality

Unit Number	Level Number	Unit Name
1	1	SATM System
A	1.1	Device Sense & Control
D	1.1.1	Door Sense & Control
2	1.1.1.1	Get Door Status
3	1.1.1.2	Control Door
4	1.1.1.3	Dispense Cash
E	1.1.2	Slot Sense & Control
5	1.1.2.1	WatchCardSlot
6	1.1.2.2	Get Deposit Slot Status
7	1.1.2.3	Control Card Roller
8	1.1.2.4	Control Envelope Roller
9	1.1.2.5	Read Card Strip
10	1.2	Central Bank Comm.
11	1.2.1	Get PIN for PAN
12	1.2.2	Get Account Status
13	1.2.3	Post Daily Transactions
B	1.3	Terminal Sense & Control
14	1.3.1	Screen Driver
15	1.3.2	Key Sensor
C	1.4	Manage Session
16	1.4.1	Validate Card
17	1.4.2	Validate PIN
18	1.4.2.1	GetPIN
F	1.4.3	Close Session
19	1.4.3.1	New Transaction Request
20	1.4.3.2	Print Receipt
21	1.4.3.3	Post Transaction Local
22	1.4.4	Manage Transaction
23	1.4.4.1	Get Transaction Type
24	1.4.4.2	Get Account Type
25	1.4.4.3	Report Balance
26	1.4.4.4	Process Deposit
27	1.4.4.5	Process Withdrawal

SATM Units and Abbreviated Names



SATM functional decomposition tree

- The decomposition tree is the basis of integration testing. It is important to remember that such a decomposition is primarily a packaging partition of the system.

- As software design moves into more detail, the added information. The functional decomposition tree into a unit calling graph.

- The Unit calling graph is the directed graph in which nodes are program units and edges runs from node A to node B.

- Drawing a call graphs do not scale up well.

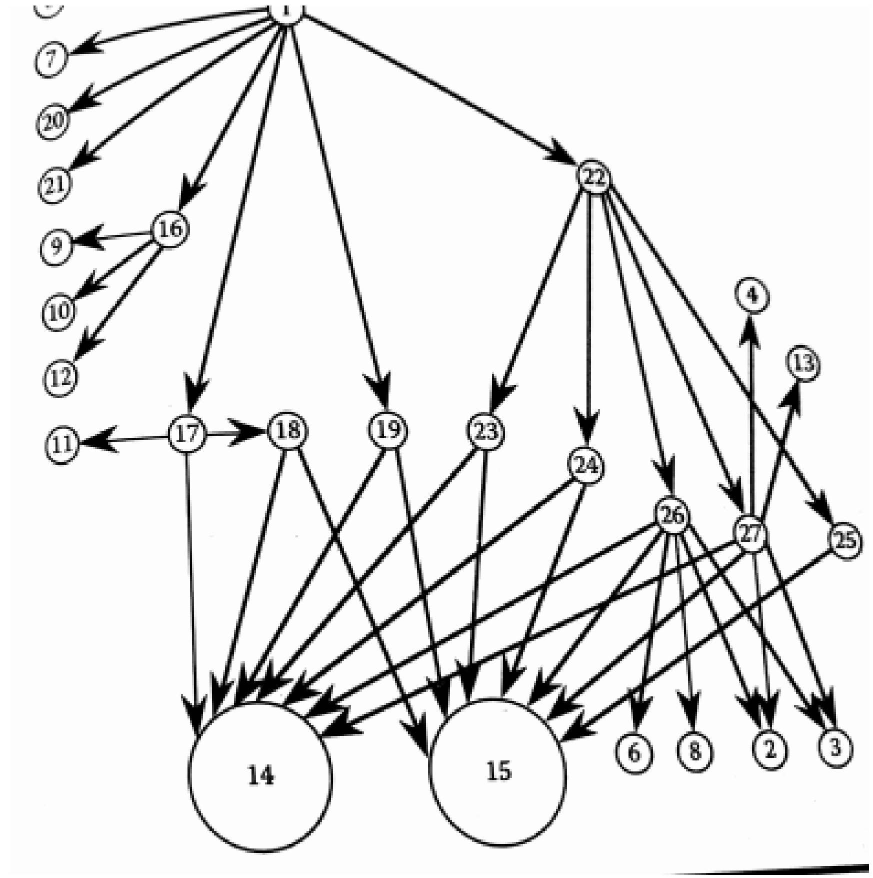
- Both the drawings and the adjacency matrix provide insights to the tester.

- Node with a higher degree will be important to integration testing and paths from the main program(node-1) to the sink nodes can be used to identify contents of builds for an incremental development.

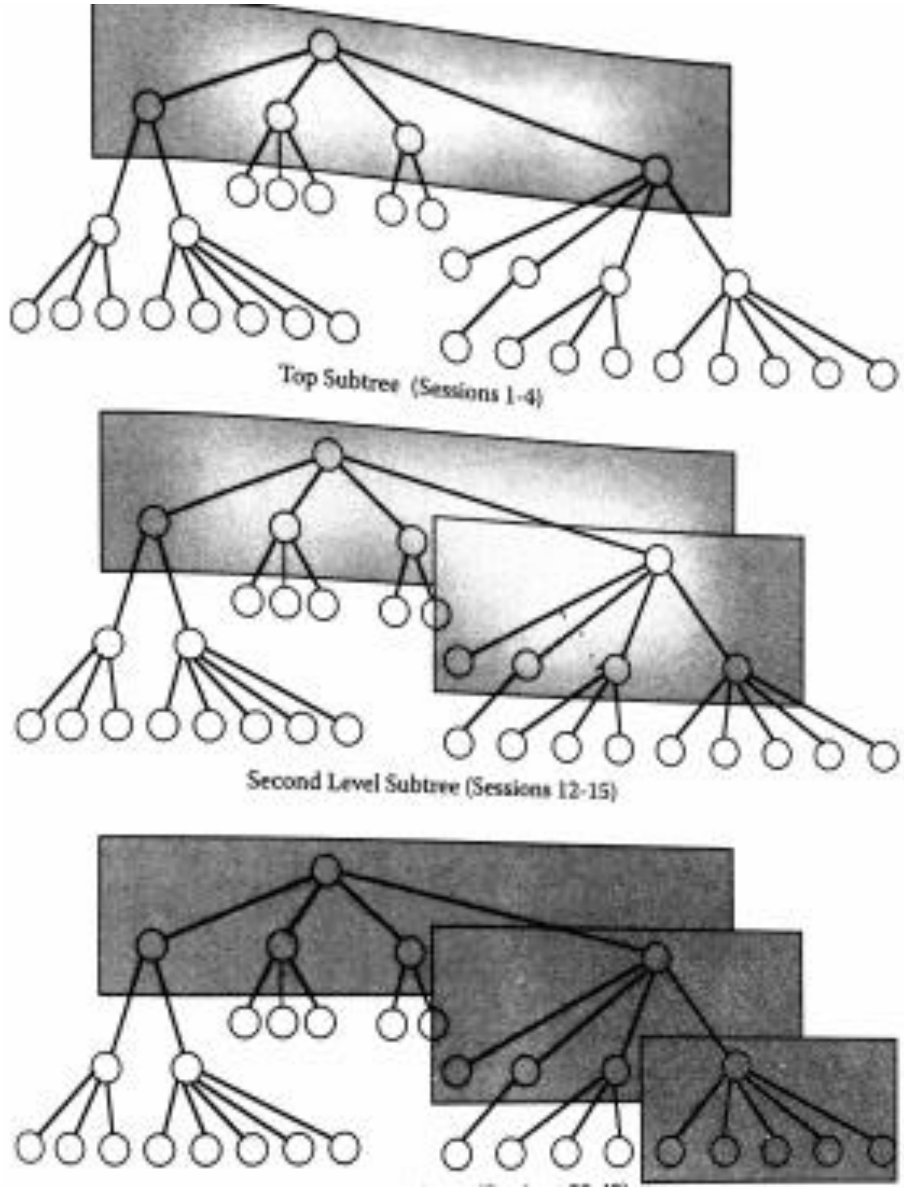
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1				X		X									X	X		X	X	X	X					
2																										
3																										
4																										
5																										
6																										
7																										
8																										
9																										
10																										
11																										
12																										
13																										
14																										
15																										
16								X	X		X															
17										X			X				X									
18													X	X												
19													X	X												
20																										
21																										
22																						X	X			
23													X	X												
24													X	X												
25														X												
26	X	X			X		X						X	X												
27	X	X	X		X		X					X	X	X												

Adjacency Matrix for the SATM Call Graph

SATM Call graph is shown in the graph. Some of the hierarchy is obscured to reduce the confusion in the drawing.



SATM Call Graph

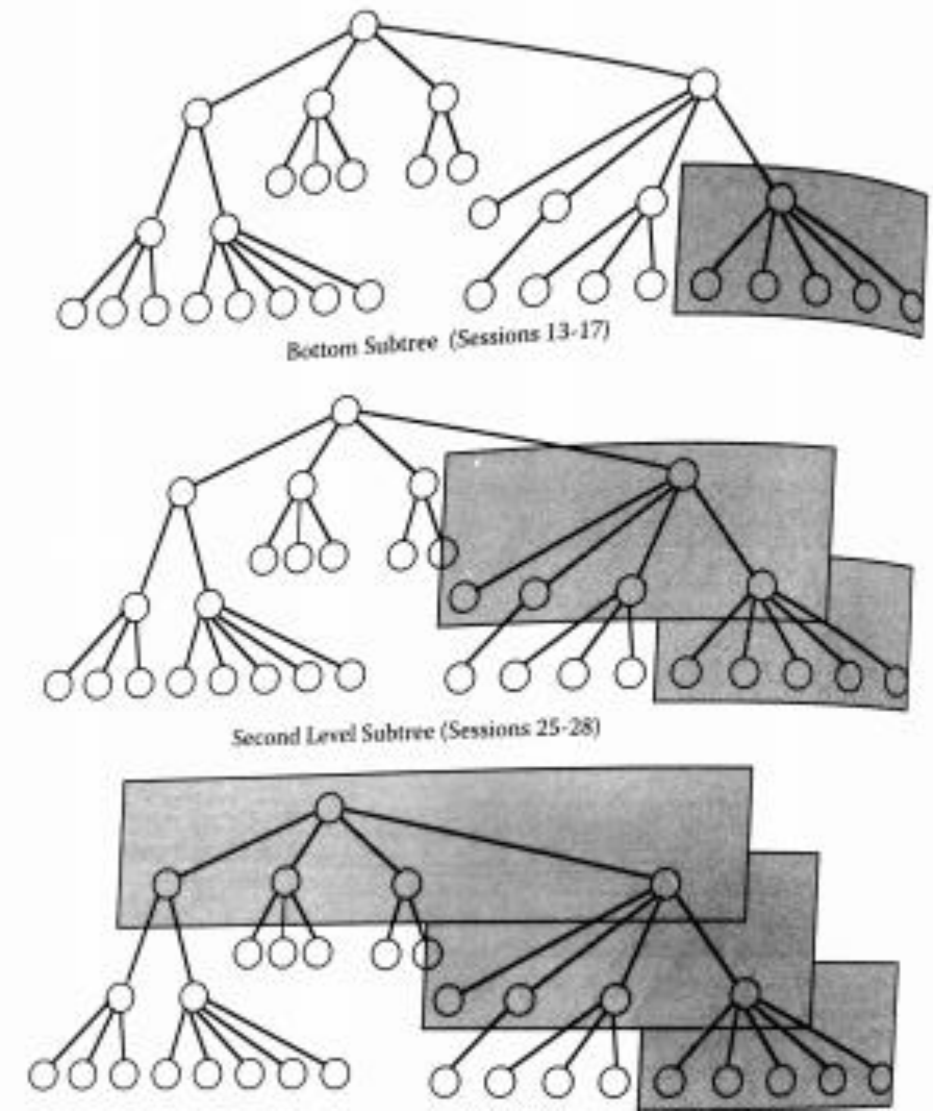


Top-Down Integration

At the uppermost level, we would have stubs for the four components in the first level decomposition.

There would be four integration sessions, in each one component would be actual code and other three would be stubs.

Top-down integration follows a breadth-first traversal of the functional decomposition tree.



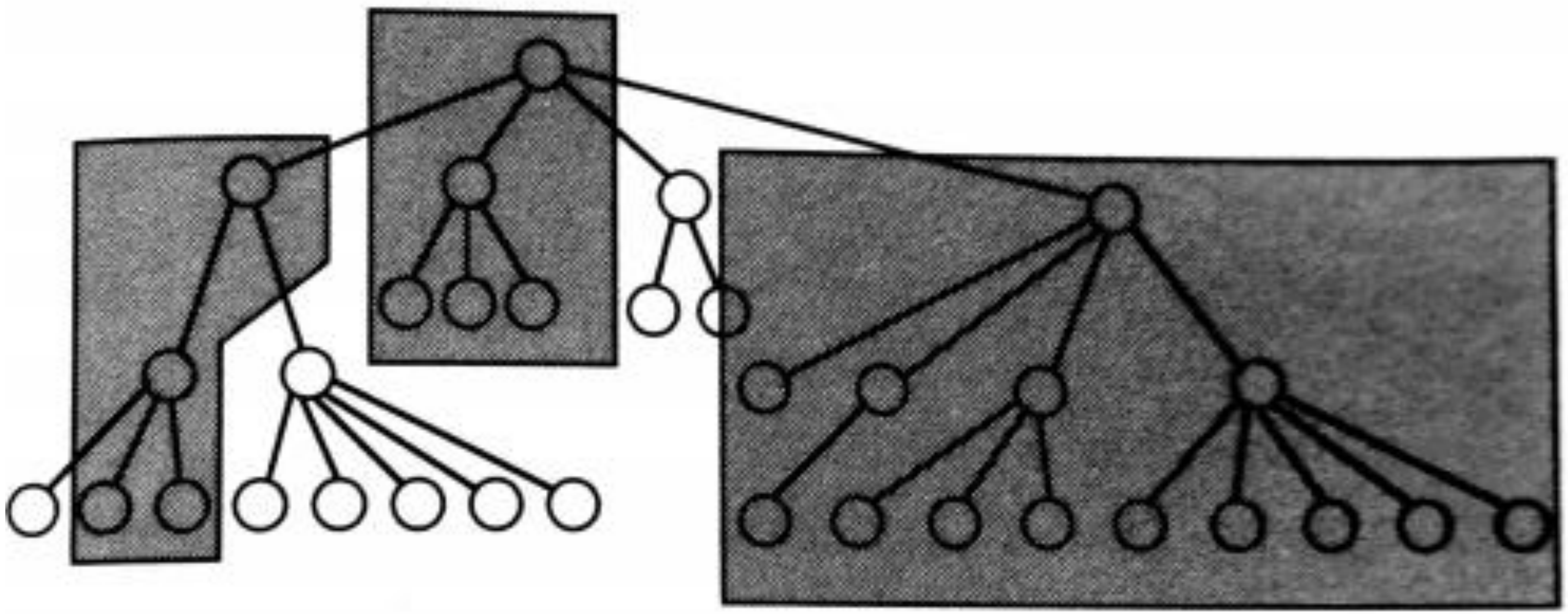
Bottom Up Integration

Bottom-up integration is a “mirror image” to the top-down order, with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree.

In bottom-up integration, we start with the leaves of the decomposition tree (units like ControlDoor and DispenseCash), and test them with specially coded drivers.

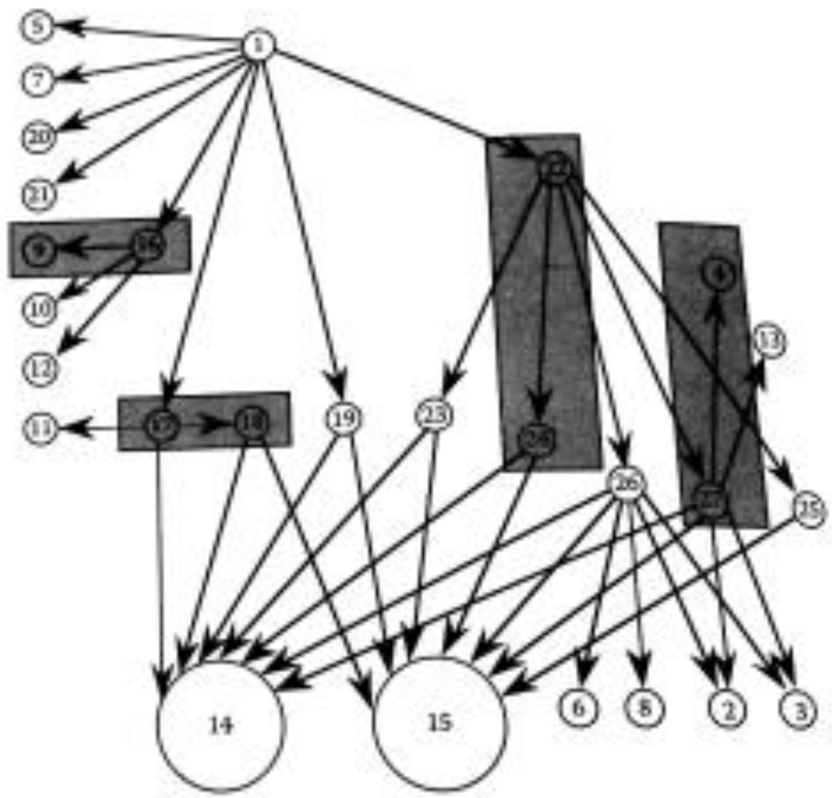
There is probably less throw-away code in drivers than there is in stubs. Recall we had one stub for each child node in the decomposition tree.

Most systems have a fairly high fan-out near at the leaves, so in the bottom-up integration order, we won’t have as many drivers. This is partially offset by the fact that the driver modules will be more complicated



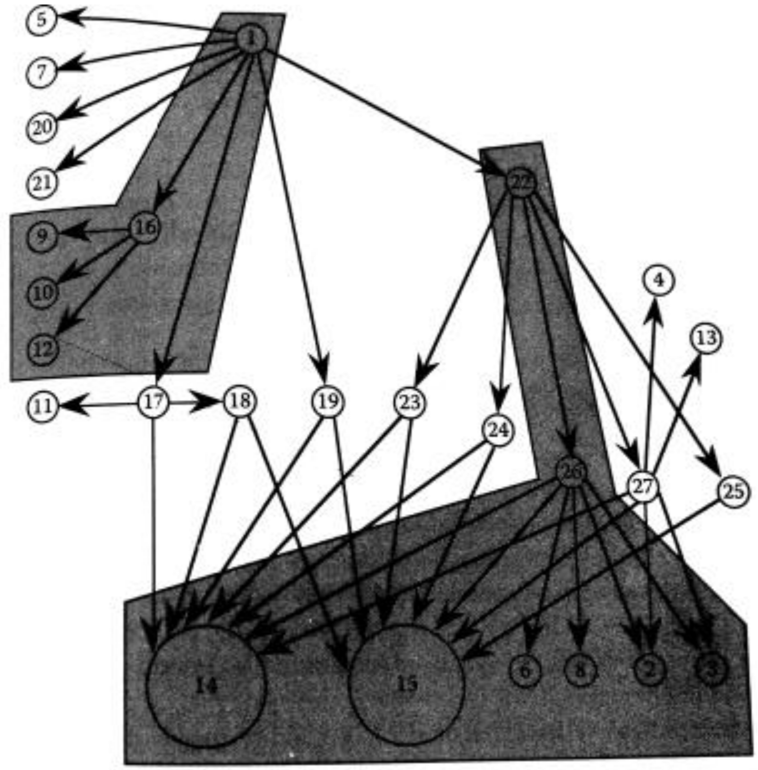
Bottom-up Integration

The idea behind pair-wise integration is to eliminate the stub/driver development effort. Rather than develop tubs and/or drivers, why not use the actual code? At first, this sounds like big bang integration, but we restrict a session to just a pair of units in the call graph. The end result is that we have one integration test session for each edge in the call graph



Pairwise Integration

We can let the mathematics carry us still further by borrowing the notion of a "neighborhood" from topology. (This isn't too much of a stretch - graph theory is a branch of topology.) We (informally) define the neighborhood of a node in a graph to be the set of nodes that are one edge away from the given node. In a directed graph, this means all the immediate predecessor nodes and all the immediate successor nodes (notice that these correspond to the set of stubs and drivers of the node).

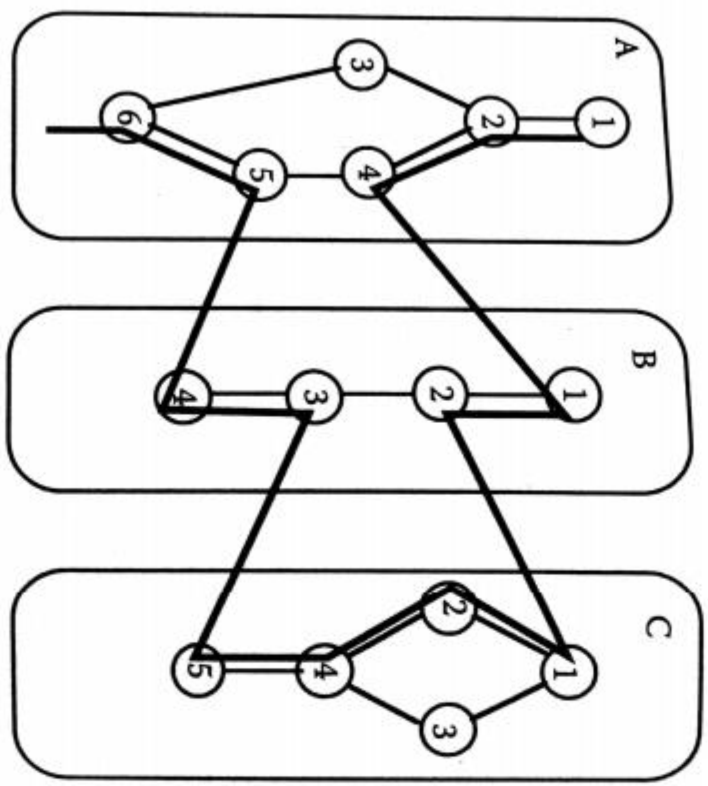


Neighbourhoods Integration

The eleven neighborhoods for the SATM example (based on the call graph in Figure 4.2) are given in Table 3.

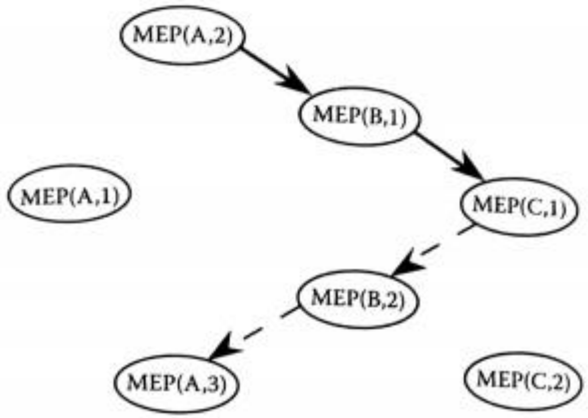
Node	Predecessors	Successors
16	1	9, 10, 12
17	1	11, 14, 18
18	17	14, 15
19	1	14, 15
23	22	14, 15
24	22	14, 15
26	22	14, 15, 6, 8, 2, 3
27	22	14, 15, 2, 3, 4, 13
25	22	15
22	1	23, 24, 26, 27, 25
1	n/a	5, 7, 2, 21, 16, 17, 19, 22

calling unit to the called unit, where some other path of source statements is traversed. We cleverly ignored this situation in Part III, because this is a better place to address the question. There are two possibilities: abandon the singleentry, single exit precept and treat such calls as an exit followed by an entry, or “suppress” the call statement because control eventually returns to the calling unit anyway. The suppression choice works well for unit testing, but it is antithetical to integration testing.



MM-Path across three units

The first guideline for MM-Paths: points of quiescence are “natural” endpoints for an MM-Path. Our second guideline also serves to distinguish integration from system testing.
Our second guideline: atomic system functions are an upper limit for MM-Paths: we don’t want MMPaths to cross ASF boundaries. This means that ASFs represent the seam between integration and system testing. They are the largest item to be tested by integration testing, and the smallest item for system testing. We can test an ASF at both levels. Again, the digit entry ASF is a good example.
During system testing, the port input event is a physical key press that is detected by KeySensor and sent to GetPIN as a string variable. (Notice that KeySensor performs the physical to logical transition.) GetPIN determines whether a digit key or the cancel key was pressed, and responds accordingly.
(Notice that button presses are ignored.) The ASF terminates with either screen 2 or 4 being displayed. Rather than require system keystrokes and visible screen displays, we could use a driver to provide these, and test the digit entry ASF via integration testing. We can see this using our continuing example.



MM-Path graph derived from previous MM-Path

Conclusion

In a nut shell we have seen a brief Integration Testing Strategies, Testing Components and assemblies, System Testing, Acceptance Testing, Regression Testing, Usability Testing, Regression Testing Selection Techniques, Test Case prioritization and Selective Execution, Levels of Testing and Integration Testing, Traditional view of testing levels, Alternative life cycle models, The SATM System, Separating Integration and System Testing, A Closer look at the SATM System, Decomposition Based, Call Graph Based and Path Based Integrations.