Software Metrics

Software Metrics

- Pressman explained as “A measure provides a quantitative indication of the amount, dimension, capacity, or size of some attribute of the product or process”.
- Measurement is the act of determine a measure

- Software metrics can be defined as “*The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products*”.

Types of Software Metrics

Measurement can be categorized in two ways :-

1. Direct Measures
2. Indirect Measures

Direct measures of the software engineering process include cost and effort applied. It includes LOC produced, execution speed, memory size, and defects reported over some set period of time.

Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability etc.

Categories of Software Metrics

i. **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

ii. **Process metrics:** describe the effectiveness and quality of the processes that produce the software product. Examples are:

- effort required in the process
- time to produce the product
- effectiveness of defect removal during development
- number of defects found during testing
- maturity of the process

iii. **Project metrics:** describe the project characteristics and execution. Examples are :

- number of software developers
- staffing pattern over the life cycle of the software
- cost and schedule
- productivity

Software Metrics: What and Why?

- *Software metric* is defined as a quantitative measure of an attribute a software system possesses with respect to Cost, Quality, Size and Schedule.
  Example-
  Measure - No. of Errors
  Metrics - No. of Errors found per person

- These are numerical data related to software development. Metrics strongly support software project management activities.

Halstead Metrics

- It is an analytical technique to measure **size, development effort and development cost of software products**.

- Halstead used a few primitive program parameters to develop the expressions for overall program length, potential minimum value, actual volume, effort and development time.

Token Count

The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

where $\eta = \eta_1 + \eta_2$

η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

Halstead Metrics

The length of the program in the terms of the total number of tokens used is

$N = N_1 + N_2$

where

N : program length

N_1 : total occurrences of operators

N_2 : total occurrences of operands

Halstead Metrics

Volume

$$V = N * \log_2 \eta$$

The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used.

Program Level

$$L = V^* / V$$

The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size).

Halstead Metrics

Program Difficulty

$$D = 1 / L$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

Effort

$$E = V / L = D * V$$

The unit of measurement of E is elementary mental discriminations.

Advantages of Halstead Metrics

- Simple to calculate
- Do not require in-depth analysis of programming structure.
- Measure overall quality of programs.
- Predicts maintenance efforts.
- Useful in scheduling the projects.

Software Metrics

- Estimated Program Length

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

$$\hat{N} = 14 \log_2 14 + 10 \log_2 10$$

= 53.34 + 33.22 = 86.56

Software Metrics

- Potential Volume

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$$

- Estimated Program Level / Difficulty

Halstead offered an alternate formula that estimate the program level.

$$\hat{L} = 2\eta_2 / (\eta_1 \mathrm{N}_2)$$

where

$$\hat{D} = \frac{1}{\hat{L}} = \frac{\eta_1 \mathrm{N}_2}{2\eta_2}$$

Software Metrics

- Effort and Time

$$\mathrm{E} = V / \hat{L} = V * \hat{D}$$

$$= (n_1 N_2 N \log_2 \eta) / 2\eta_2$$

$$T = E / \beta$$

β is normally set to 18 since this seemed to give best results in Halstead's earliest experiments, which compared the predicted times with observed programming times, including the time for design, coding, and testing.

Software Metrics

- Language Level

$$\lambda = L \times V^* = L^2 V$$

Using this formula, Halstead and other researchers determined the language level for various languages as shown in Table 1.

Software Metrics

Language	*Language Level* λ	*Variance* σ
PL/1	1.53	0.92
ALGOL	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42
PASCAL	2.54	–
APL	2.42	–
C	0.857	0.445

Table 1: Language levels

Counting rules for C language

1. Comments are not considered.

2. All the variables and constants are considered operands.

3. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.
4. Local variables with the same name in different functions are counted as unique operands.

5. Functions calls are considered as operators.

6. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.

7. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.

8. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.

9. All the brackets, commas, and terminators are considered as operators.

10. GOTO is counted as an operator and the label is counted as an operand.

11. The unary and binary occurrence of “+” and “-” are dealt separately. Similarly “*” (multiplication operator) are dealt with separately.

12. In the array variables such as “array-name [index]” “array-name” and “index” are considered as operands and [ ] is considered as operator.

13. In the structure variables such as “struct-name, member-name” or “struct-name -> member-name”, struct-name, member-name are taken as operands and ‘.’, ‘->’ are taken as operators. Some names of member elements in different structure variables are counted as unique operands.

14. All the hash directive are ignored.

Example

```
z=0;
while x>0
        z=z+y;
        x=x-1;
end while;
print (z);
```

Operators	Count	Operands	Count
=	3	z	4
;	5	x	3
while-end while	1	y	1
>	1	0	2
+	1	1	1
-	1	-	-
print	1	-	-
()	1	-	-

Here, $\eta_1 = 8,\ \eta_2 = 5,\ N_1 = 14,\ N_2 = 11$

Halstead's Software Metrics

- Program Length (N) = N1+N2 = 25
- Program volume (V) = N * $\log_2 \eta$ = 92.51
- Program Length Equation N` = $\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ = 35.61
- Potential Volume(V*) = $(2 + \eta_2^*) \log_2 (2 + \eta_2^*)$ = 8
- Program length(N) = N Program Level (L) = V*/V = 0.086
- Program Difficulty (D) = 1/L = 11.56
- Programming Effort (E) = V/L = D*V = 1069.76
- Time (T') = E/18 = 59 sec

Example

```
int f=1, n=7;
for (int i=1; i<=n; i+=1)
f*=i;
```

int	2	f	2
=	3	1	3
,	1	n	2
;	4	7	1
for	1	i	4
()	1	-	-
<=	1	-	-
+=	1	-	-
*=	1	-	-
9	15	5	12

Solution

N, the length of the program to be N1+N2 = 27

V, the volume of the program, to be N × $\log_2(\eta_1 + \eta_2)$ = 102.8

Halstead's Software Metrics

- Program Length (N) = N1+N2 = 27
- Program volume (V) = N * $\log_2 \eta$ = 102.8
- Potential Volume(V*) = $(2 + \eta_2^*) \log_2 (2 + \eta_2^*)$ = 2
- Program length(N) = N Program Level (L) = V*/V = 0.019
- Program Difficulty (D) = 1/L = 51.4
- Programming Effort (E) = V/L = D*V = 5283.92
- Time (T') = E/18 = 293 sec

Example

Consider the sorting program. List out the operators and operands and also calculate the values of software science measures like η, N, V, E $etc.$

```
int sort (int x[ ], int n)
{
        int i,j,save,im1;
        // This function sorts array x in ascending order
        if(n < 2) return 1;
        for (i = 2; i <=n; i++)
        {
                im1 = i-1;
                for( j =1; j<=im1; j++)
                        if( x[i] < x[j])
                        {
                                save = x[i];
                                x[i] = x[j];
                                x[j] = save;
                        }
        }
        return 0;
}
```

Solution

The list of operators and operands is given in table.

Operators	*Occurrences*	*Operands*	*Occurrences*
int	4	SORT	1
( )	5	*x*	7
,	4	*n*	3
[ ]	7	*i*	8
if	2	*j*	7
<	2	save	3

(Contd.)...

;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
< =	2	—	—
+ +	2	—	—
return	2	—	—
{ }	3	—	—
$\eta_1 = 14$	$N_1 = 53$	$\eta_2 = 10$	$N_2 = 38$

Table : Operators and operands of sorting program

Here N_1=53 and N_2=38. The program length N=N_1+N_2=91

Vocabulary of the program $\eta = \eta_1 + \eta_2 = 14 + 10 = 24$

Volume $V = N \times \log_2 \eta$

= 91 x $\log_2$24 = 417 bits

The estimated program length $\hat{N}$ of the program

= 14 $\log_2$14 + 10 $\log_2$10

= 14 * 3.81 + 10 * 3.32

= 53.34 + 33.2 = 86.45

Conceptually unique input and output parameters are represented by η_2^*

$\eta_2^* = 3$ {x: array holding the integer to be sorted. This is used both as input and output}.

{N: the size of the array to be sorted}.

The potential volume V* = 5 $\log_2 5$ = 11.6

Since L = V* / V

$$= \frac{11.6}{417} = 0.027$$

D = I / L

$$= \frac{1}{0.027} = 37.03$$

Effort Equation

$$E = V / L = 15450.08$$

Therefore, 15450 elementary mental discrimination are required to construct the program.

Time Equation

$$T = E / 18 = \frac{15450}{18} = 858 \text{ seconds} = 14 \text{ minutes}$$

This is probably a reasonable time to produce the program, which is very simple

Example

Consider the program shown in table. Calculate the various software science metrics.

```
#include < stdio.h >
#define MAXLINE 100
int getline(char line[],int max);
int strindex(char source[],char search for[]);
char pattern[ ]="ould";
int main()
{
     char line[MAXLINE];
     int found = 0;
     while(getline(line,MAXLINE)>0)
          if(strindex(line, pattern)>=0)
          {
               printf("%s",line);
               found++;
          }
     return found;
}
```

(Contd.)...

```
int getline(char s[],int lim)
{
     int c,i=0;
     while(--lim > 0 && (c=getchar())!= EOF && c!='\n')
          s[i++]=c;
     if(c=='\n')
          s[i++] = c;
     s[i] = '\0';
     return i;
}
int strindex(char s[],char t[])
{
     int i,j,k;
     for(i=0;s[i]  !='\0';i++)
     {
          for(j=i,k=0;t[k]  !=  '\0',s[j]  ==t[k];j++,k++);
          if(k>0 && t[k] =='\0')
               return i;
     }
     return -1;
}
```

Solution

List of operators and operands are given in Table below.

Operators	*Occurrences*	*Operands*	*Occurrences*
main ()	1	—	—
–	1	**Extern variable** pattern	1
for	2	**main function** line	3
= =	3	found	2
! =	4	**getline function** s	3
getchar	1	lim	1

(Contd.)...

( )	1	*c*	5
&&	3	*i*	4
– –	1	**Strindex function** *s*	2
return	4	*t*	3
+ +	6	*i*	5
printf	1	*j*	3
> =	1	*k*	6
strindex	1	**Numerical Operands** 1	1
If	3	MAXLINE	1
>	3	0	8
getline	1	'\0'	4
while	2	'\n'	2
{ }	5	**strings "ould"**	1
=	10	—	—
[ ]	9	—	—
,	6	—	—
;	14	—	—
EOF	1	—	—
$\eta_1 = 24$	$N_1 = 84$	$\eta_2 = 18$	$N_2 = 55$

Program vocabulary $\eta = 42$

Program length $N = N_1 + N_2$

$= 84 + 55 = 139$

Estimated length $\hat{N} = 24\log_2 24 + 18\log_2 18 = 185.115$

% error = 24.91

Program volume V = 749.605 bits

Estimated program level $= \frac{2}{\eta_1} \times \frac{\eta_2}{N_2}$

$= \frac{2}{24} \times \frac{18}{55} = 0.02727$

Minimal volume $V^* = 20.4417$

Effort $= V / \hat{L}$

$$= \frac{748.605}{.02727}$$

= 27488.33 elementary mental discriminations.

Time T = $E / \beta = \frac{27488.33}{18}$

= 1527.1295 seconds

= 25.452 minutes

Software Reliability

Software Reliability

Basic Concepts

There are three phases in the life of any hardware component i.e., burn-in, useful life & wear-out.

In **burn-in phase,** failure rate is quite high initially, and it starts decreasing gradually as the time progresses.

During **useful life period**, failure rate is approximately constant.

Failure rate increase in **wear-out phase** due to wearing out/aging of components. The best period is useful life period. The shape of this curve is like a “bath tub” and that is why it is known as bath tub curve. The “bath tub curve” is given in Fig.7.1.

Software Reliability

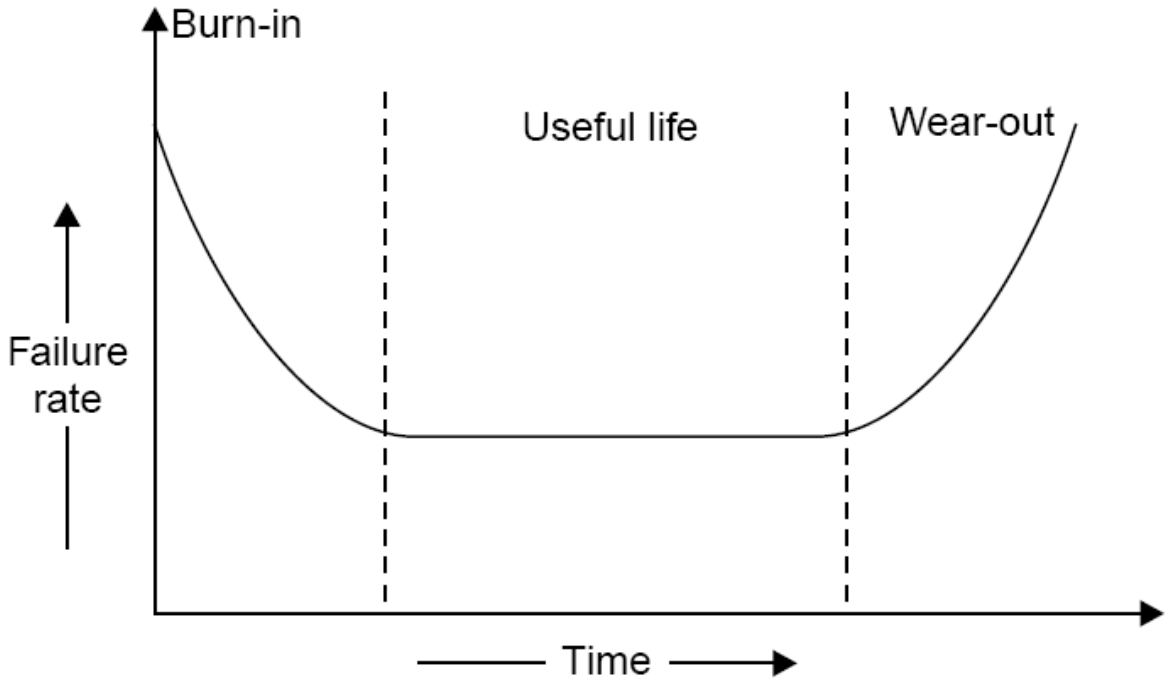


Fig. 7.1: Bath tub curve of hardware reliability.

Software Reliability

We do not have wear out phase in software. The expected curve for software is given in fig. 7.2.

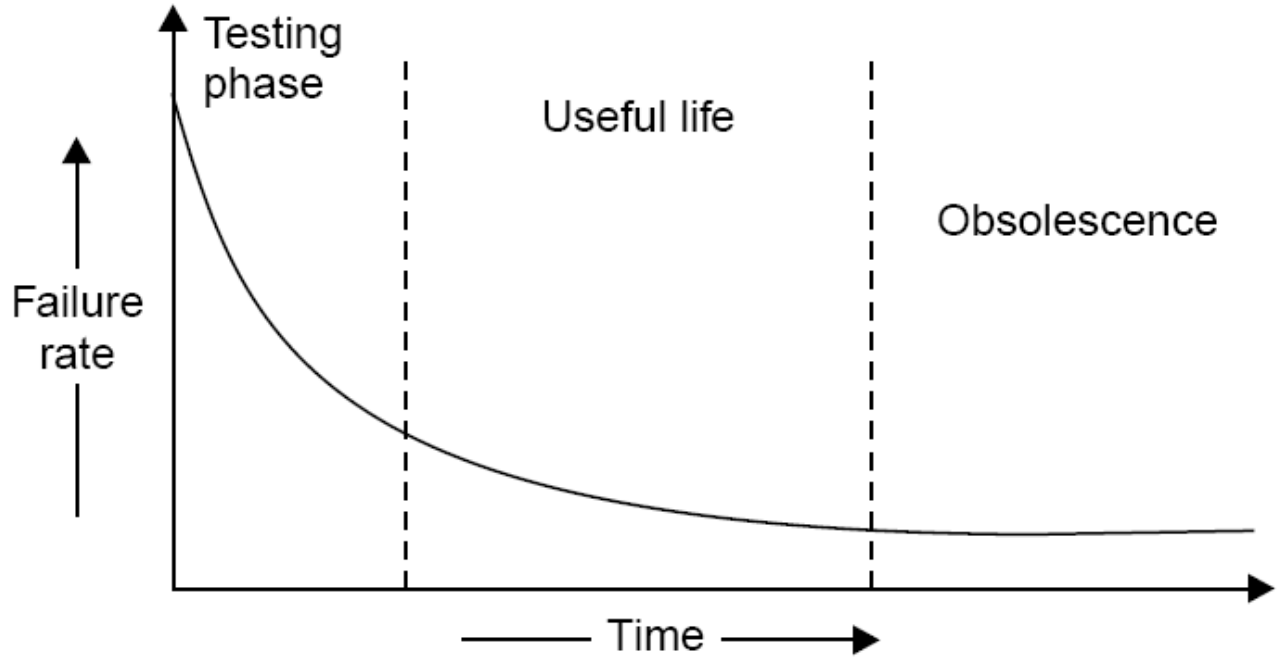


Fig. 7.2: Software reliability curve (failure rate versus time)

Software Reliability

Software may be retired only if it becomes obsolete. Some of contributing factors are given below:

- ✓ change in environment
- ✓ change in infrastructure/technology
- ✓ major change in requirements
- ✓ increase in complexity
- ✓ extremely difficult to maintain
- ✓ deterioration in structure of the code
- ✓ slow execution speed
- ✓ poor graphical user interfaces

Software Reliability

What is Software Reliability?

"Software reliability means operational reliability. Who cares how many bugs are in the program?

As per IEEE standard: "Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time".

Software Reliability

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the inputs are free of error.

"It is the probability of a failure free operation of a program for a specified time in a specified environment".

Software Reliability

- Failures and Faults

A fault is the defect in the program that, when executed under particular conditions, causes a failure.

The execution time for a program is the time that is actually spent by a processor in executing the instructions of that program. The second kind of time is calendar time. It is the familiar time that we normally experience.

Software Reliability

There are four general ways of characterising failure occurrences in time:

1. time of failure,
2. time interval between failures,
3. cumulative failure experienced up to a given time,
4. failures experienced in a time interval.

Software Reliability

Uses of Reliability Studies

There are at least four other ways in which software reliability measures can be of great value to the software engineer, manager or user.

1. you can use software reliability measures to evaluate software engineering technology quantitatively.

2. software reliability measures offer you the possibility of evaluating development status during the test phases of a project.

Software Reliability

3. one can use software reliability measures to monitor the operational performance of software and to control new features added and design changes made to the software.

4. a quantitative understanding of software quality and the various factors influencing it and affected by it enriches into the software product and the software development process.

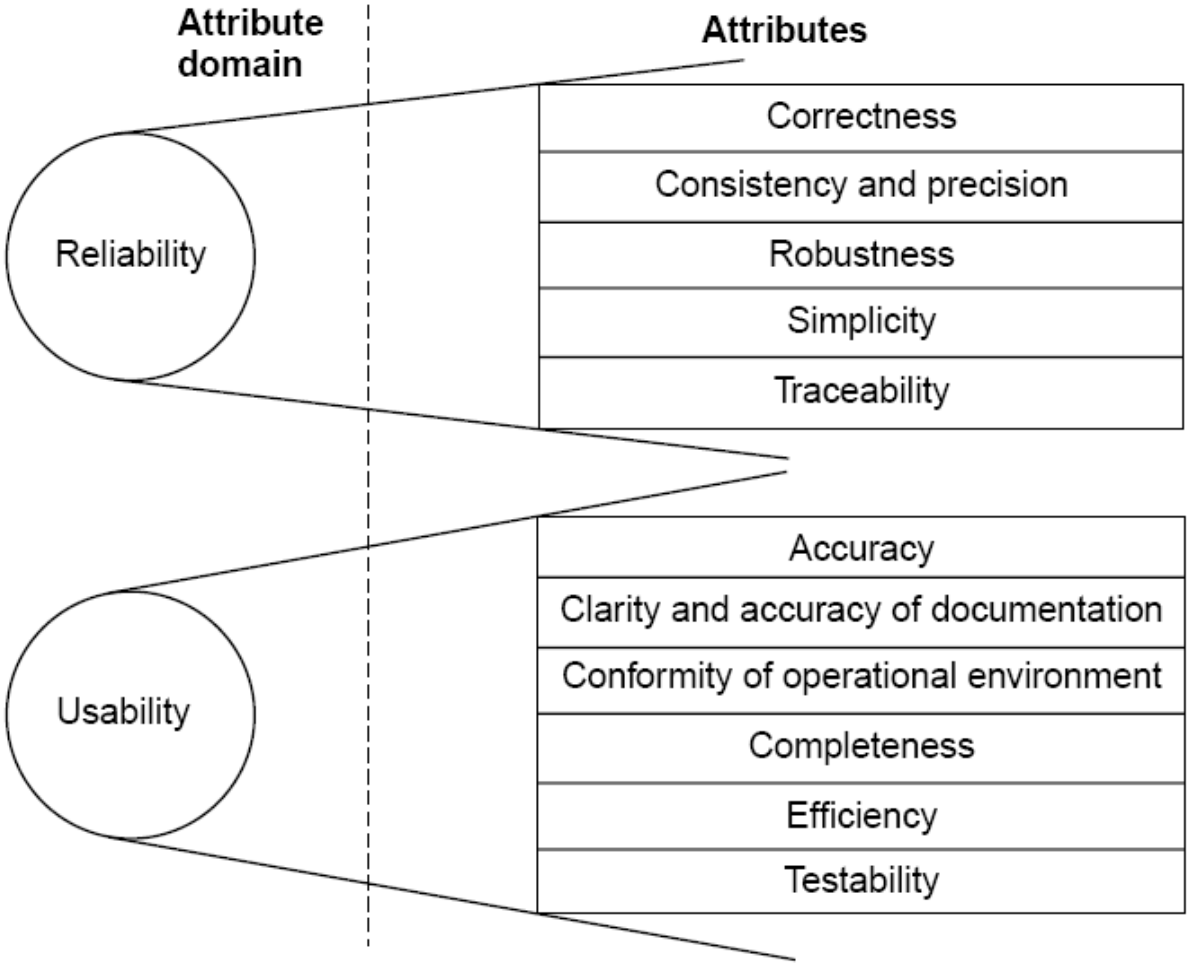
Software Reliability

Software Quality

Different people understand different meanings of quality like:

- conformance to requirements
- fitness for the purpose
- level of satisfaction

Software Reliability



Software Reliability

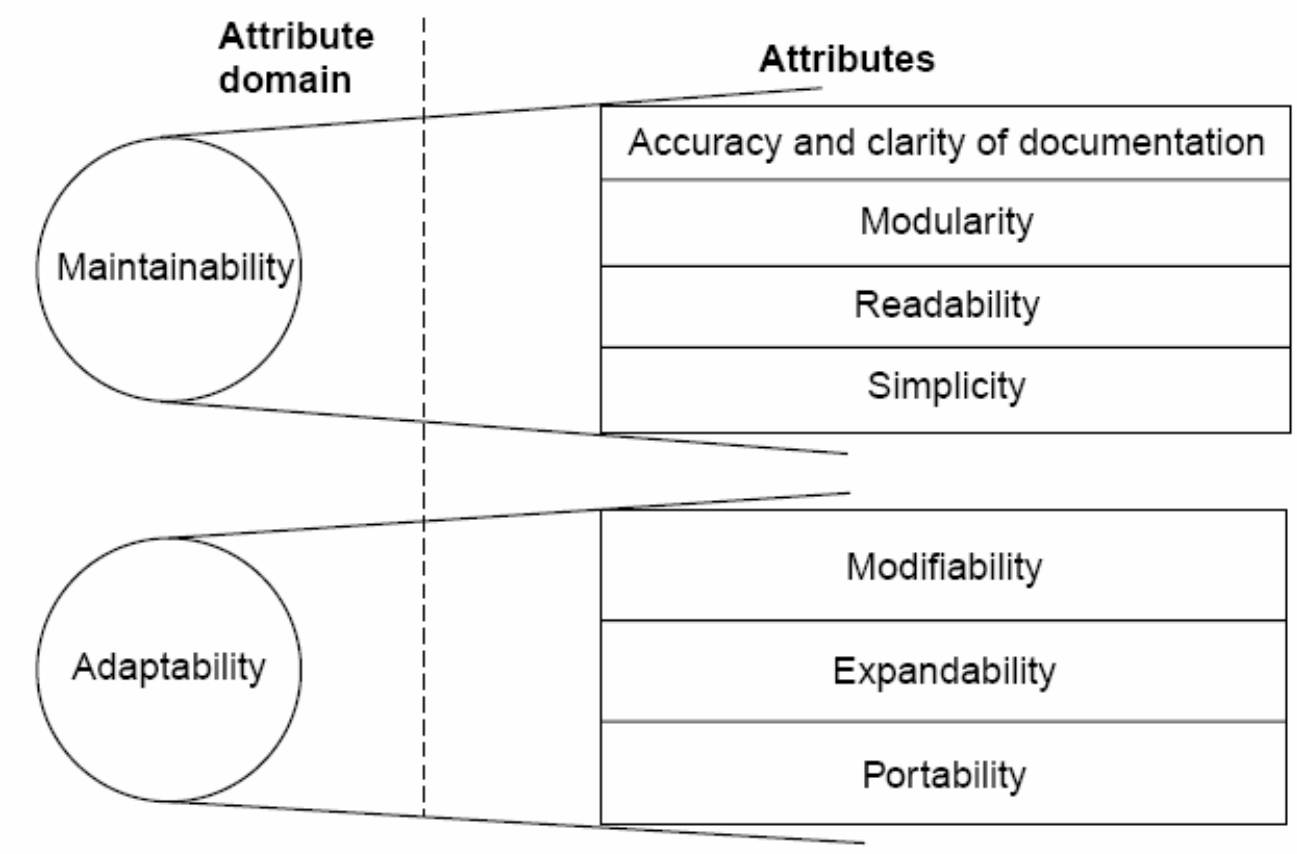


Fig 7.8: Software quality attributes

Software Reliability

1	Reliability	The extent to which a software performs its intended functions without failure.
2	Correctness	The extent to which a software meets its specifications.
3	Consistency & precision	The extent to which a software is consistent and give results with precision.
4	Robustness	The extent to which a software tolerates the unexpected problems.
5	Simplicity	The extent to which a software is simple in its operations.
6	Traceability	The extent to which an error is traceable in order to fix it.
7	Usability	The extent of effort required to learn, operate and understand the functions of the software

(Contd.)...

Software Reliability

8	Accuracy	Meeting specifications with precision.
9	Clarity & Accuracy of documentation	The extent to which documents are clearly & accurately written.
10	Conformity of operational environment	The extent to which a software is in conformity of operational environment.
11	Completeness	The extent to which a software has specified functions.
12	Efficiency	The amount of computing resources and code required by software to perform a function.
13	Testability	The effort required to test a software to ensure that it performs its intended functions.
14	Maintainability	The effort required to locate and fix an error during maintenance phase.

(Contd.)...

Software Reliability

15	Modularity	It is the extent of ease to implement, test, debug and maintain the software.
16	Readability	The extent to which a software is readable in order to understand.
17	Adaptability	The extent to which a software is adaptable to new platforms & technologies.
18	Modifiability	The effort required to modify a software during maintenance phase.
19	Expandability	The extent to which a software is expandable without undesirable side effects.
20	Portability	The effort required to transfer a program from one platform to another platform.

Table 7.4: Software quality attributes

Software Reliability

- McCall Software Quality Model

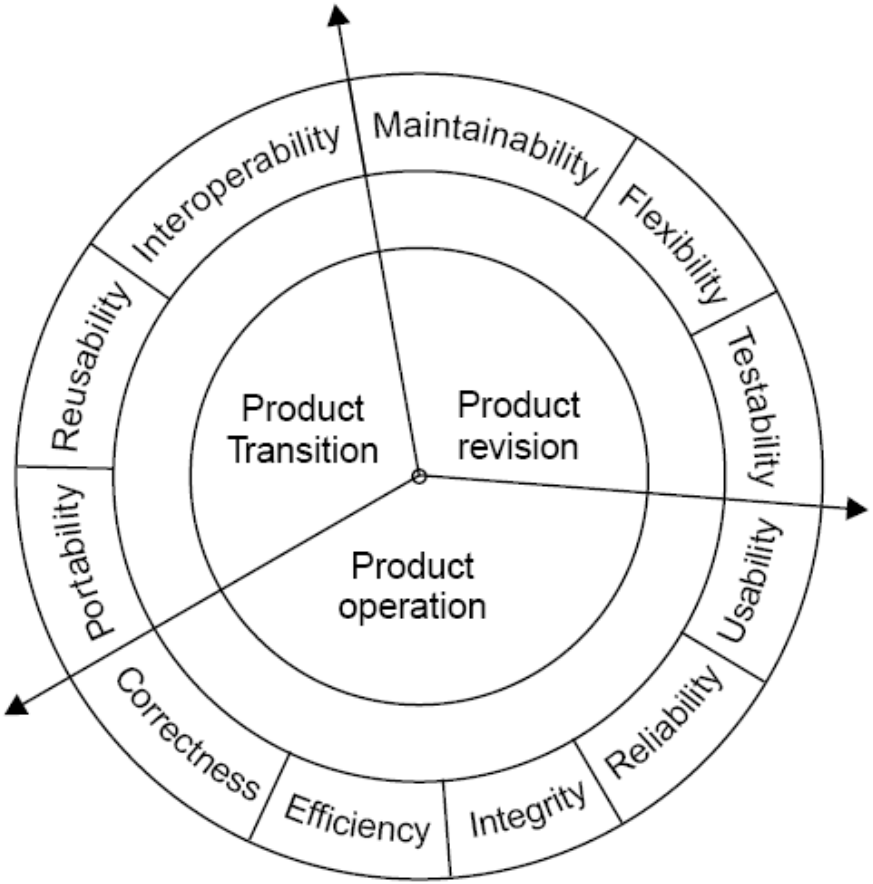


Fig 7.9: Software quality factors

Software Reliability

i. Product Operation

Factors which are related to the operation of a product are combined. The factors are:

- Correctness
- Efficiency
- Integrity
- Reliability
- Usability

These five factors are related to operational performance, convenience, ease of usage and its correctness. These factors play a very significant role in building customer's satisfaction.

Software Reliability

ii. Product Revision

The factors which are required for testing & maintenance are combined and are given below:

- Maintainability
- Flexibility
- Testability

These factors pertain to the testing & maintainability of software. They give us idea about ease of maintenance, flexibility and testing effort. Hence, they are combined under the umbrella of product revision.

Software Reliability

iii. Product Transition

We may have to transfer a product from one platform to an other platform or from one technology to another technology. The factors related to such a transfer are combined and given below:

- Portability
- Reusability
- Interoperability

Software Reliability

Most of the quality factors are explained in table 7.4. The remaining factors are given in table 7.5.

Sr.No.	Quality Factors	Purpose
1	Integrity	The extent to which access to software or data by the unauthorized persons can be controlled.
2	Flexibility	The effort required to modify an operational program.
3	Reusability	The extent to which a program can be reused in other applications.
4	Interoperability	The effort required to couple one system with another.

Table 7.5: Remaining quality factors (other are in table 7.4)

Quality criteria

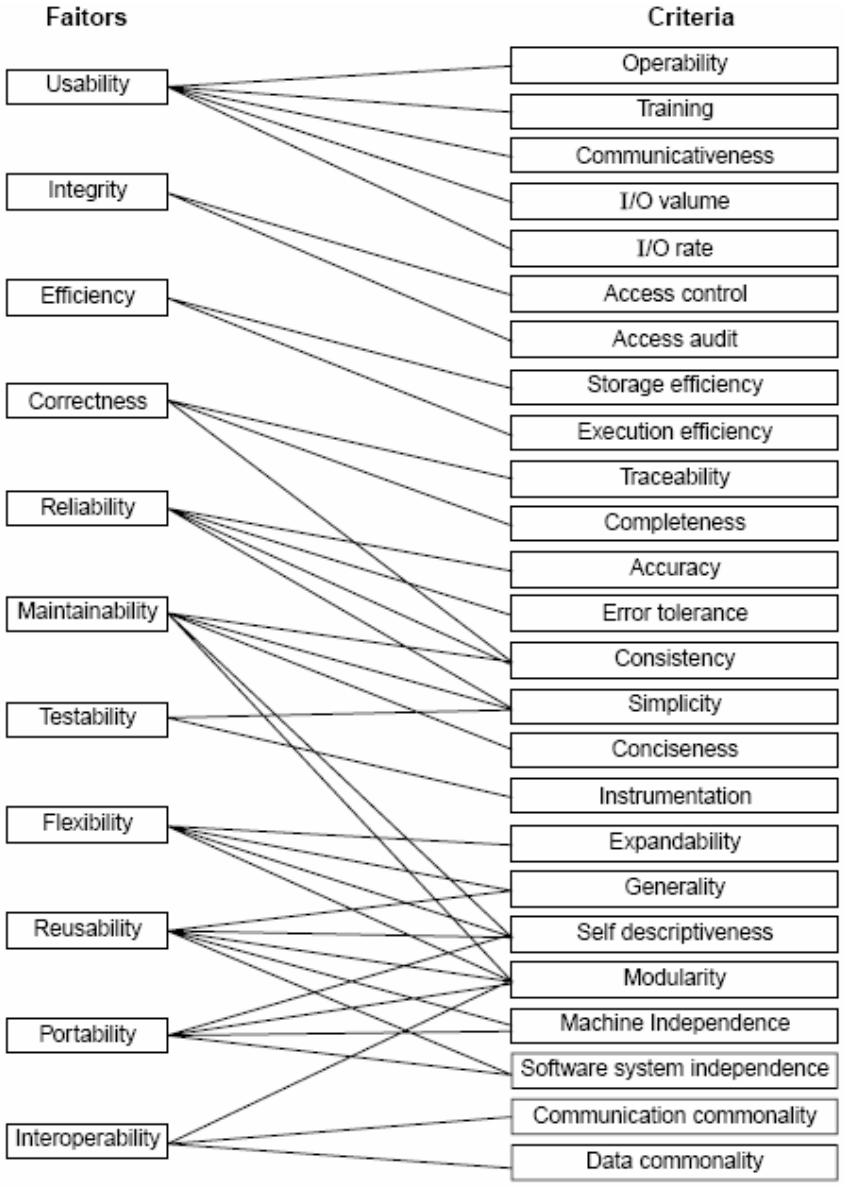


Fig 7.10: McCall's quality model

Software Reliability

Sr. No.	Quality Criteria	Usability	Integrity	Efficiency	Correctness	Reliability	Maintainability	Testability	Flexibility	Reusability	Portability	Interoperability
1.	Operability	×										
2.	Training	×										
3.	Communicativeness	×										
4.	I/O volume	×										
5.	I/O rate	×										
6.	Access control		×									
7.	Access Audit		×									
8.	Storage efficiency			×								
9.	Execution Efficiency			×								
10.	Traceability				×							
11.	Completeness				×							
12.	Accuracy					×						
13.	Error tolerance					×						
14.	Consistency				×	×	×					
15.	Simplicity					×	×	×				
16.	Conciseness						×					
17.	Instrumentation							×				
18.	Expandability								×			
19.	Generality								×	×		
20.	Self-descriptiveness						×		×	×	×	
21.	Modularity						×		×	×	×	×
22.	Machine independence									×	×	
23.	S/W system independence									×	×	
24.	Communication commonality											×
25.	Data commonality											×

Table 7.5(a): Relation between quality factors and quality criteria

Software Reliability

1	Operability	The ease of operation of the software.
2	Training	The ease with which new users can use the system.
3	Communicativeness	The ease with which inputs and outputs can be assimilated.
4	I/O volume	It is related to the I/O volume.
5	I/O rate	It is the indication of I/O rate.
6	Access control	The provisions for control and protection of the software and data.
7	Access audit	The ease with which software and data can be checked for compliance with standards or other requirements.
8	Storage efficiency	The run time storage requirements of the software.
9	Execution efficiency	The run-time efficiency of the software.

(Contd.)...

Software Reliability

10	Traceability	The ability to link software components to requirements.
11	Completeness	The degree to which a full implementation of the required functionality has been achieved.
12	Accuracy	The precision of computations and output.
13	Error tolerance	The degree to which continuity of operation is ensured under adverse conditions.
14	Consistency	The use of uniform design and implementation techniques and notations throughout a project.
15	Simplicity	The ease with which the software can be understood.
16	Conciseness	The compactness of the source code, in terms of lines of code.
17	Instrumentation	The degree to which the software provides for measurements of its use or identification of errors.

(Contd.)...

Software Reliability

18	Expandability	The degree to which storage requirements or software functions can be expanded.
19	Generability	The breadth of the potential application of software components.
20	Self-descriptiveness	The degree to which the documents are self explanatory.
21	Modularity	The provision of highly independent modules.
22	Machine independence	The degree to which software is dependent on its associated hardware.
23	Software system independence	The degree to which software is independent of its environment.
24	Communication commonality	The degree to which standard protocols and interfaces are used.
25	Data commonality	The use of standard data representations.

Table 7.5 (b): Software quality criteria

Software Reliability

- Boehm Software Quality Model

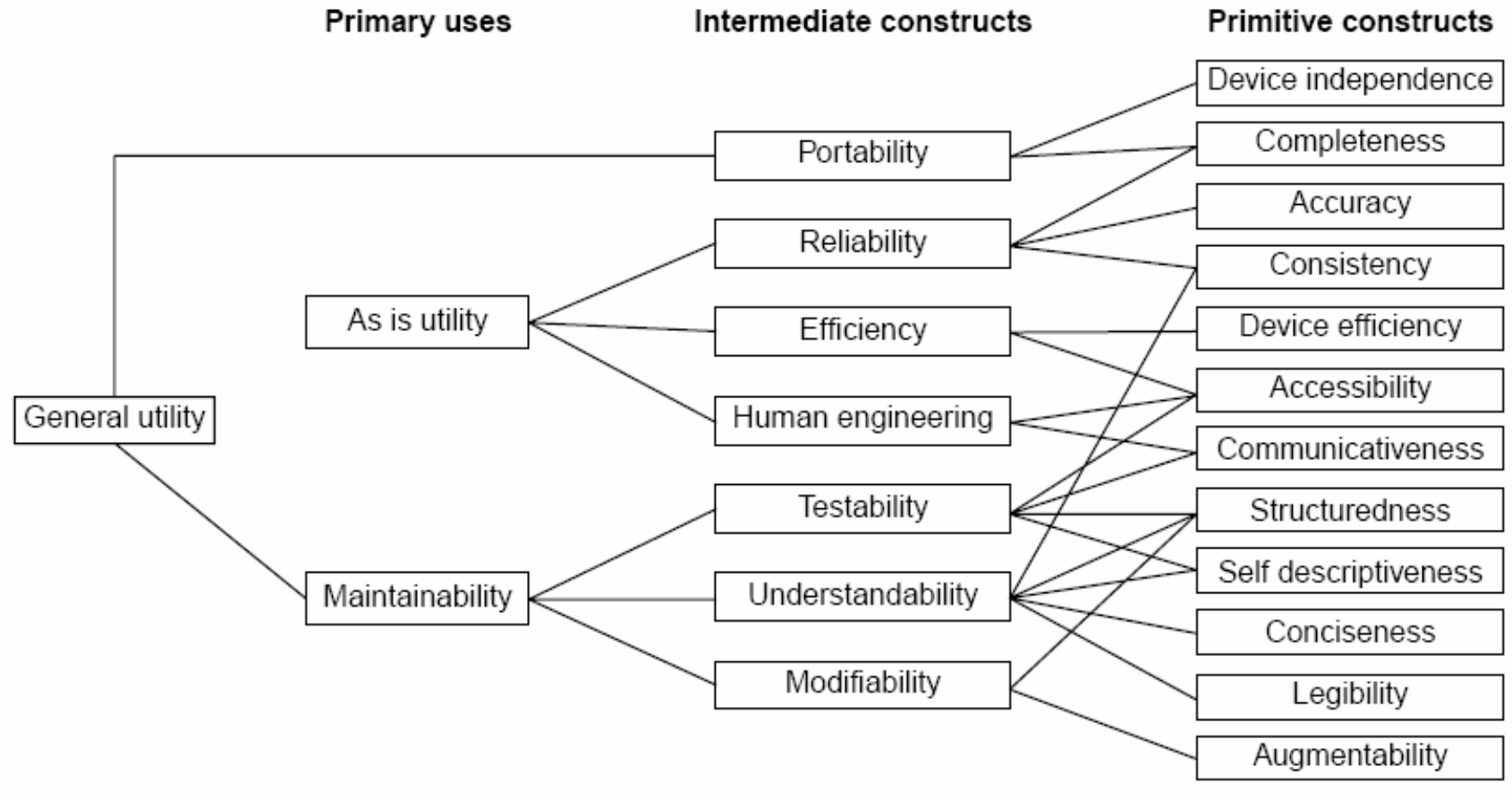
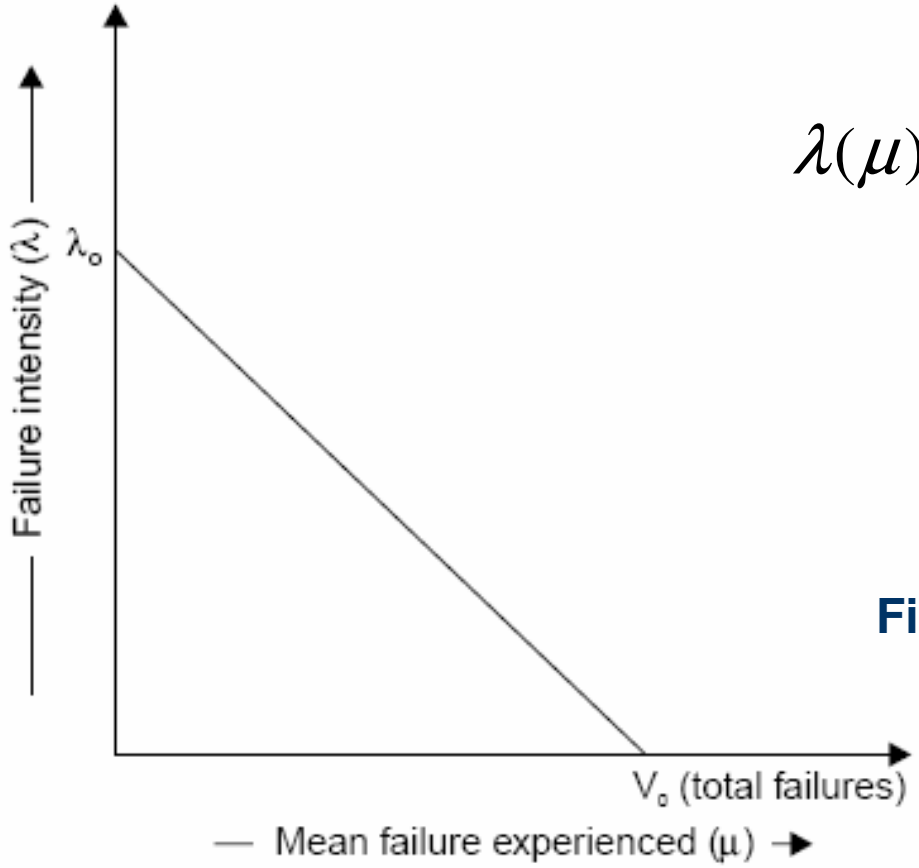


Fig.7.11: The Boehm software quality model

Software Reliability Models

- Basic Execution Time Model



$$\lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{V_0}\right) \qquad (1)$$

Fig.7.13: Failure intensity λ as a function of μ for basic model

Software Reliability

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} \qquad (2)$$

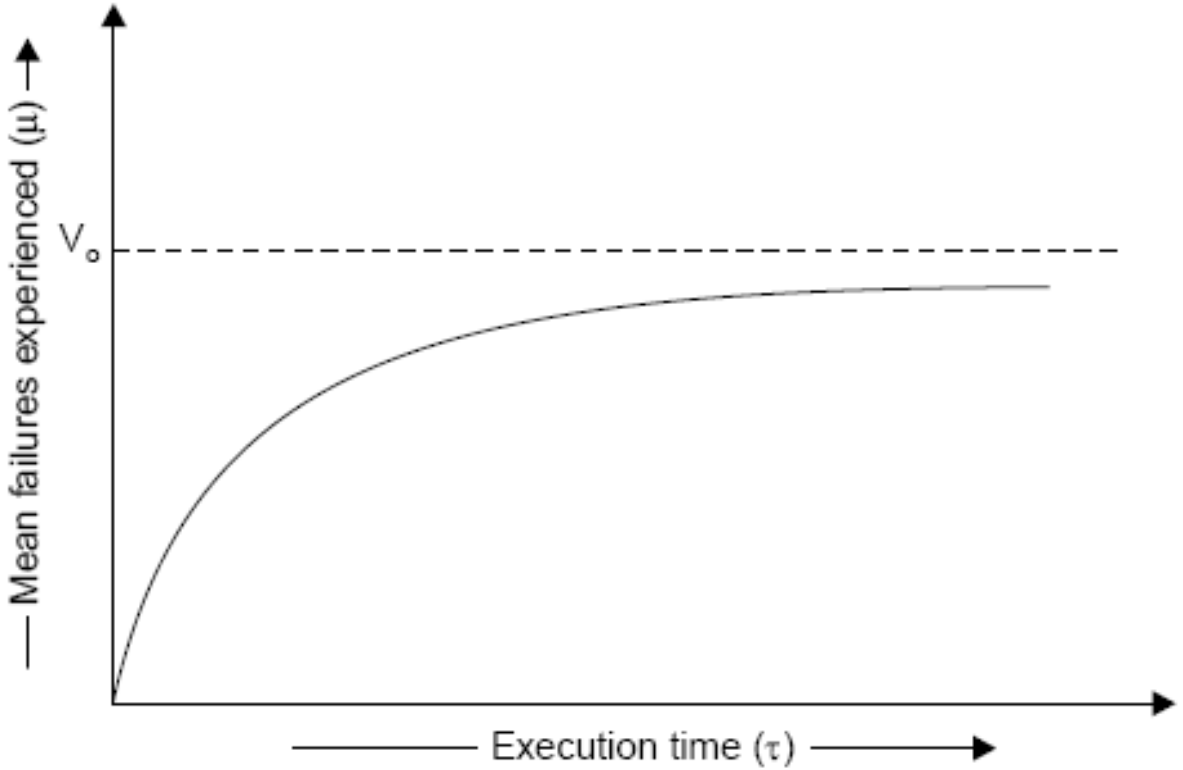


Fig.7.14: Relationship between τ & μ for basic model

Software Reliability

For a derivation of this relationship, equation 1 can be written as:

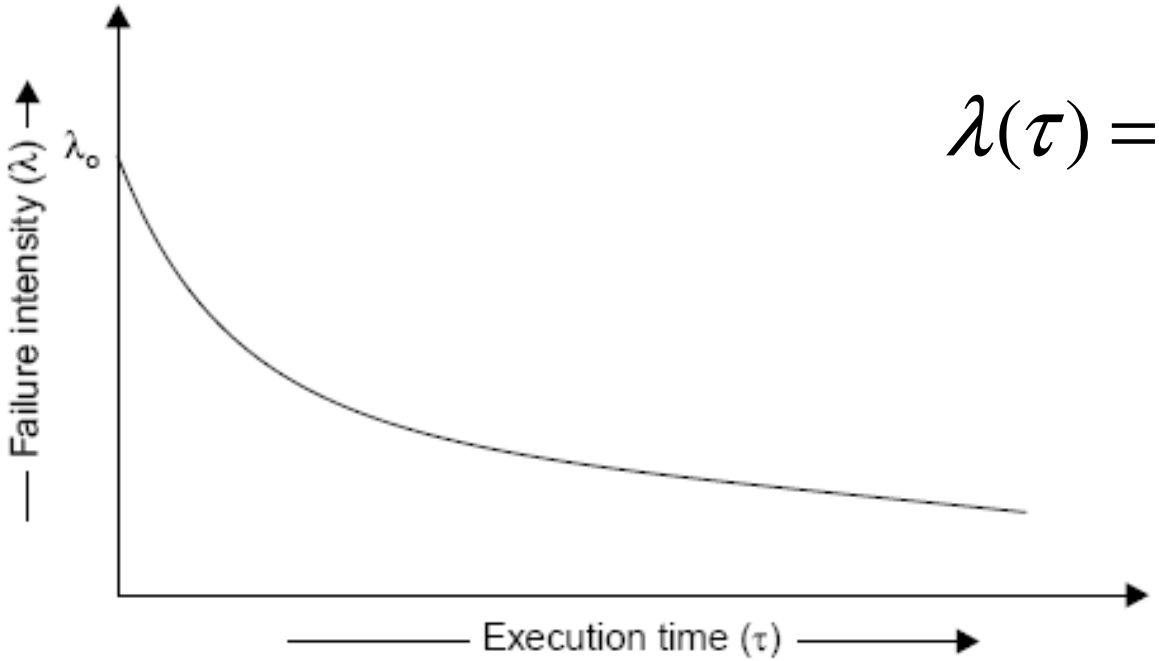
$$\frac{d\mu(\tau)}{d\tau} = \lambda_0\left(1 - \frac{\mu(\tau)}{V_0}\right)$$

The above equation can be solved for $\mu(\tau)$ and result in :

$$\mu(\tau) = V_0\left(1 - \exp\left(\frac{-\lambda_0\tau}{V_0}\right)\right) \quad \textbf{(3)}$$

Software Reliability

The failure intensity as a function of execution time is shown in figure given below



$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)$$

Fig.7.15: Failure intensity versus execution time for basic model

Software Reliability

- Derived quantities

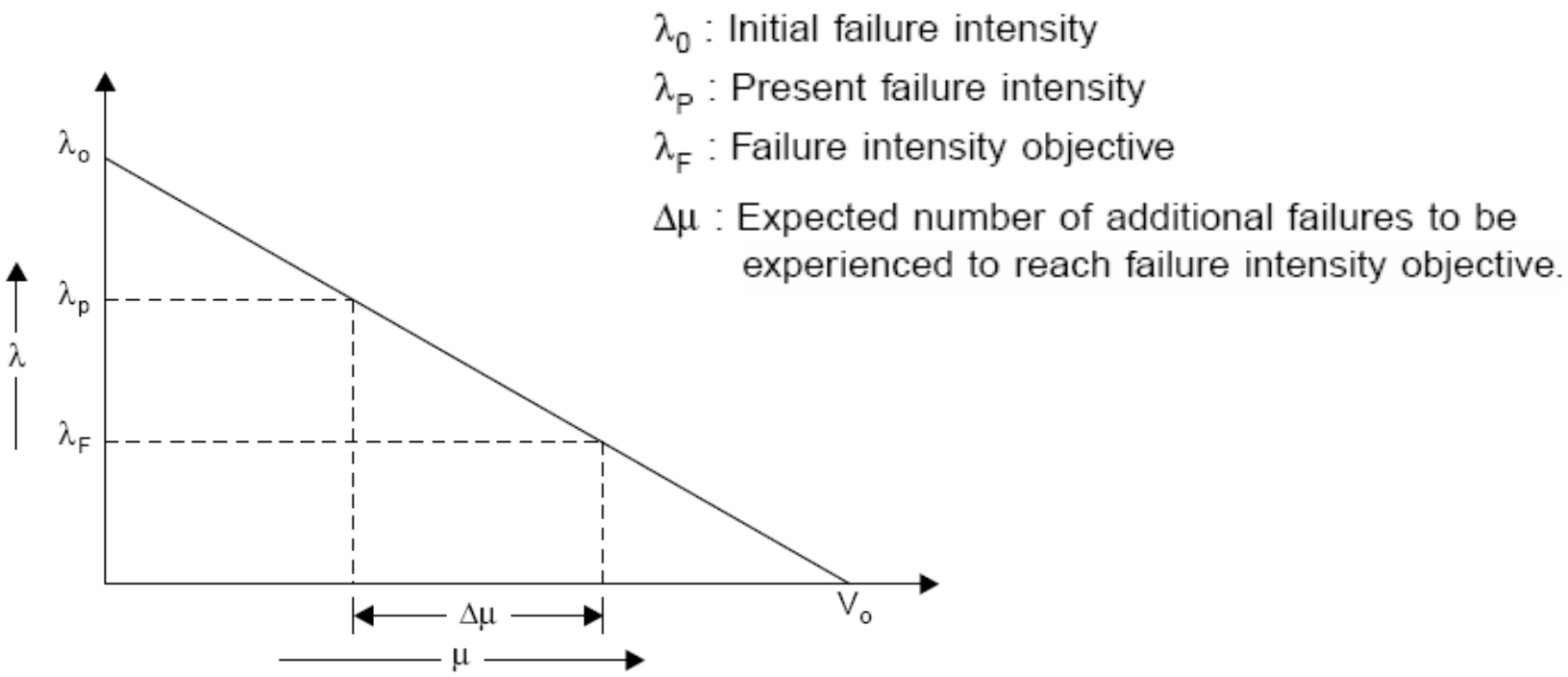


Fig.7.16: Additional failures required to be experienced to reach the objective

Software Reliability

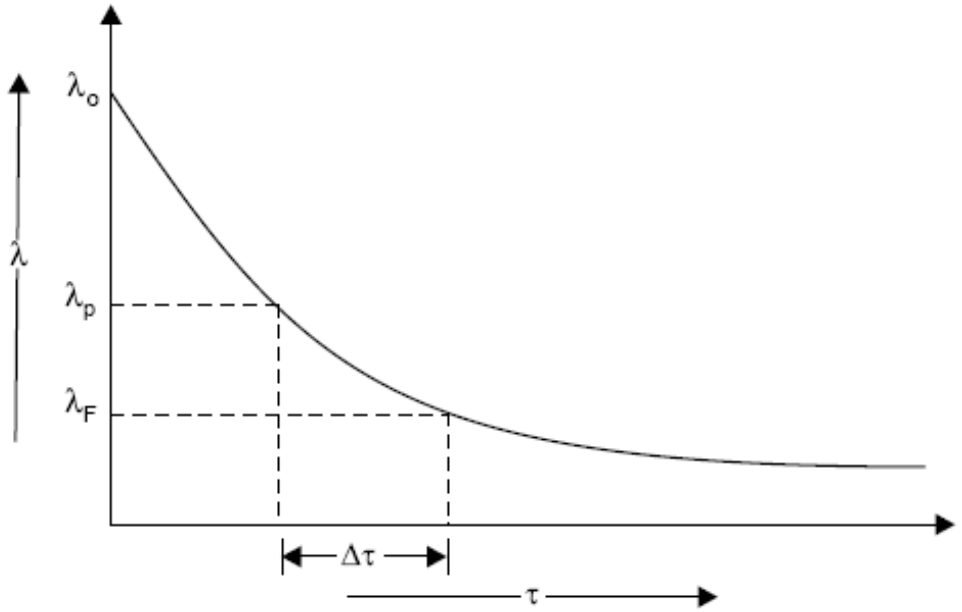


Fig.7.17: Additional time required to reach the objective

This can be derived in mathematical form as:

$$\Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

Software Reliability

Example- 7.1

Assume that a program will experience 200 failures in infinite time. It has now experienced 100. The initial failure intensity was 20 failures/CPU hr.

(i) Determine the current failure intensity.

(ii) Find the decrement of failure intensity per failure.

(iii) Calculate the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.

(iv) Compute addition failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr.

Use the basic execution time model for the above mentioned calculations.

Solution

Here V_o=200 failures

$$\mu = 100 \text{ failures}$$

$$\lambda_0 = 20 \text{ failures/CPU hr.}$$

(i) Current failure intensity:

$$\lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{V_0}\right)$$

$$= 20\left(1 - \frac{100}{200}\right) = 20(1 - 0.5) = 10 \text{ failures/CPU hr}$$

(ii) Decrement of failure intensity per failure can be calculated as:

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} = -\frac{20}{200} = -0.1/\text{CPU hr.}$$

(iii) (a) Failures experienced & failure intensity after 20 CPU hr:

$$\mu(\tau) = V_0\left(1 - \exp\left(\frac{-\lambda_0\tau}{V_0}\right)\right)$$

$$= 200\left(1 - \exp\left(\frac{-20\times 20}{200}\right)\right) = 200(1 - \exp(1-2))$$

$$= 200(1 - 0.1353) \approx 173 \text{failures}$$

Software Reliability

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)$$

$$= 20 \exp\left(\frac{-20 \times 20}{200}\right) = 20 \exp(-2) = 2.71 \, failures / CPU \ \ hr$$

(b) Failures experienced & failure intensity after 100 CPU hr:

$$\mu(\tau) = V_0\left(1 - \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)\right)$$

$$= 200\left(1 - \exp\left(\frac{-20 \times 100}{200}\right)\right) = 200 \text{ failures(almost)}$$

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)$$

$$= 20 \exp\left(\frac{-20 \times 100}{200}\right) = 0.000908 \ failures / CPU \ hr$$

(iv) Additional failures $(\Delta\mu)$ required to reach the failure intensity objective of 5 failures/CPU hr.

$$\Delta\mu = \left(\frac{V_0}{\lambda_0}\right)(\lambda_P - \lambda_F) = \left(\frac{200}{20}\right)(10-5) = 50 \text{ failures}$$

Software Reliability

Additional execution time required to reach failure intensity objective of 5 failures/CPU hr.

$$\Delta\tau = \left(\frac{V_0}{\lambda_0}\right) Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{200}{20} Ln\left(\frac{10}{5}\right) = 6.93 \text{ CPU hr.}$$

Software Reliability

Example- 7.6

A program is expected to have 500 faults. It is also assumed that one fault may lead to one failure only. The initial failure intensity was 2 failures/CPU hr. The program was to be released with a failure intensity objective of 5 failures/100 CPU hr. Calculated the number of failure experienced before release.

Software Reliability

Solution

The number of failure experienced during testing can be calculated using the equation mentioned below:

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$$

Here $V_0 = 500$ because one fault leads to one failure

$\lambda_0 = 2$ failures/CPU hr.

$\lambda_F = 5$ failures/100 CPU hr.

$= 0.05$ failures/CPU hr.

Software Reliability

So $$\Delta\mu = \frac{500}{2}(2 - 0.05)$$

= 487 failures

Hence 13 faults are expected to remain at the release instant of the software.

Software Reliability

- Capability Maturity Model

It is a strategy for improving the software process, irrespective of the actual life cycle model used.

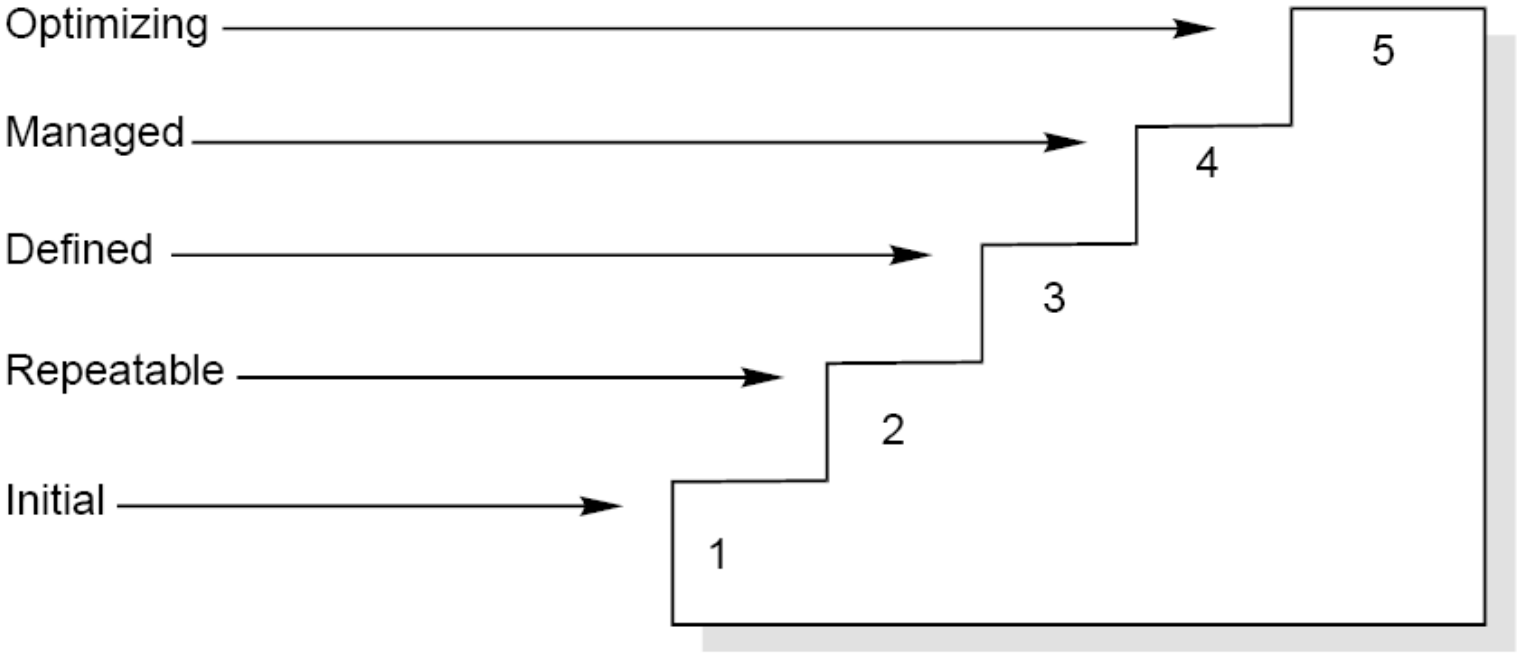


Fig.7.23: Maturity levels of CMM

Software Reliability

Maturity Levels:

- ✓ Initial (Maturity Level 1)
- ✓ Repeatable (Maturity Level 2)
- ✓ Defined (Maturity Level 3)
- ✓ Managed (Maturity Level 4)
- ✓ Optimizing (Maturity Level 5)

Software Reliability

Maturity Level	Characterization
Initial	Adhoc Process
Repeatable	Basic Project Management
Defined	Process Definition
Managed	Process Measurement
Optimizing	Process Control

Fig.7.24: The five levels of CMM

Software Reliability

- Key Process Areas

The key process areas at level 2 focus on the software project's concerns related to establishing basic project management controls, as summarized below:

Requirements Management (RM)	Establish a common relationship between the customer requirements and the developers in order to understand the requirements of the project.
Software Project Planning (PP)	Establish reasonable plans for performing the software engineering and for managing the software project.
Software Project Tracking and Oversight (PT)	Establish adequate visibility into actual progress so that management can take effective actions when the software project's performance deviates significantly from the software plans.
Software Subcontract Management (SM)	Select qualified software subcontractors and manage them effectively.
Software Quality Assurance (QA)	Provide management with appropriate visibility into the process being used by the software project and of the products being built.
Software Configuration Management (CM)	Establish and maintain the integrity of the products of the software project throughout the project's software life cycle.

Software Reliability

The key process areas at level 3 address both project and organizational issues, as summarized below:

Organization Process Focus (PF)	Establish the organizational responsibility for software process activities that improve the organization's overall software process capability.
Organization Process Definition (PD)	Develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for cumulative, long-term benefits to the organization.
Training Program (TP)	Develop the skills and knowledge of individuals so that they can perform their roles effectively and efficiently.
Integrated Software Management (IM)	Integrate the software engineering and management activities into a coherent, defined software process that is tailored from the organization's standard software process and related process assets.

(Contd.)...

Software Reliability

Software Product Engineering (PE)	Consistently perform a well-defined engineering process that integrates all the software engineering activities to produce correct, consistent software products effectively and efficiently.
Inter group Coordination (IC)	Establish a means for the software engineering group to participate actively with the other engineering groups so the project is better able to satisfy the customer's needs effectively and efficiently.
Peer Reviews (PR)	Remove defects from the software work products early and efficiently. An important corollary effect is to develop a better understanding of the software work products and of the defects that can be prevented.

Software Reliability

The key process areas at level 4 focus on establishing a quantitative understanding of both the software process and the software work products being built, as summarized below:

Quantitative Process Management (QP)	Control the process performance of the software project quantitatively.
Software Quality Management (QM)	Develop a quantitative understanding of the quality of the project's software products and achieve specific quality goals.

Software Reliability

The key process areas at level 5 cover the issues that both the organization and the projects must address to implement continuous and measurable software process improvement, as summarized below:

Defect Prevention (DP)	Identify the causes of defects and prevent them from recurring.
Technology Change Management (TM)	Identify beneficial new technologies (i.e., tools, methods, and processes) and transfer them into the organization in an orderly manner.
Process Change Management (PC)	Continually improve the software processes used in the organization with the intent of improving software quality, increasing productivity, and decreasing the cycle time for product development.

Software Reliability

- Common Features

Commitment to Perform (CO)	Describes the actions the organizations must take to ensure that the process is established and will endure. It includes practices on policy and leadership.
Ability to Perform (AB)	Describes the preconditions that must exist in the project or organization to implement the software process competently. It includes practices on resources, organizational structure, training, and tools.
Activities Performed (AC)	Describes the role and procedures necessary to implement a key process area. It includes practices on plans, procedures, work performed, tracking, and corrective action.
Measurement and Analysis (ME)	Describes the need to measure the process and analyze the measurements. It includes examples of measurements.
Verifying Implementation (VE)	Describes the steps to ensure that the activities are performed in compliance with the process that has been established. It includes practices on management reviews and audits.

Multiple Choice Questions

7.14 Which one is not a product quality factor of McCall quality model?

(a) Product revision
(b) Product operation
(c) Product specification
(d) Product transition

7.15 The second level of quality attributes in McCall quality model are termed as

(a) quality criteria
(b) quality factors
(c) quality guidelines
(d) quality specifications

7.16 Which one is not a level in Boehm software quality model ?

(a) Primary uses
(b) Intermediate constructs
(c) Primitive constructs
(d) Final constructs

7.17 Which one is not a software quality model?

(a) McCall model
(b) Boehm model
(c) ISO 9000
(d) ISO 9126

7.18 Basic execution time model was developed by

(a) Bev.Littlewood
(b) J.D.Musa
(c) R.Pressman
(d) Victor Baisili

Multiple Choice Questions

7.19 NHPP stands for

(a) Non Homogeneous Poisson Process (b) Non Hetrogeneous Poisson Process

(c) Non Homogeneous Poisson Product (d) Non Hetrogeneous Poisson Product

7.20 In Basic execution time model, failure intensity is given by

$(a)\ \lambda(\mu) = \lambda_0\left(1 - \frac{\mu^2}{V_0}\right)$ $(b)\ \lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{V_0}\right)$

$(c)\ \lambda(\mu) = \lambda_0\left(1 - \frac{V_0}{\mu}\right)$ $(d)\ \lambda(\mu) = \lambda_0\left(1 - \frac{V_0}{\mu^2}\right)$

7.21 In Basic execution time model, additional number of failures required to achieve a failure intensity objective $(\Delta\mu)$ is expressed as

$(a)\ \Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$ $(b)\ \Delta\mu = \frac{V_0}{\lambda_0}(\lambda_F - \lambda_P)$

$(c)\ \Delta\mu = \frac{\lambda_0}{V_0}(\lambda_F - \lambda_P)$ $(d)\ \Delta\mu = \frac{\lambda_0}{V_0}(\lambda_P - \lambda_F)$

Multiple Choice Questions

7.22 In Basic execution time model, additional time required to achieve a failure intensity objective $(\Delta\tau)$ is given as

$(a)\ \Delta\tau = \frac{\lambda_0}{V_0} Ln\left(\frac{\lambda_F}{\lambda_P}\right)$ $(b)\ \Delta\tau = \frac{\lambda_0}{V_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$

$(c)\ \Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_F}{\lambda_P}\right)$ $(d)\ \Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$

7.23 Failure intensity function of Logarithmic Poisson execution model is given as

$(a)\ \lambda(\mu) = \lambda_0 LN(-\theta\mu)$ $(b)\ \lambda(\mu) = \lambda_0 \exp(\theta\mu)$

$(c)\ \lambda(\mu) = \lambda_0 \exp(-\theta\mu)$ $(d)\ \lambda(\mu) = \lambda_0 \log(-\theta\mu)$

7.24 In Logarithmic Poisson execution model, 'θ' is known as

(a) Failure intensity function parameter (b) Failure intensity decay parameter

(c) Failure intensity measurement (d) Failure intensity increment parameter

Multiple Choice Questions

7.25 In jelinski-Moranda model, failure intensity is defined aseneous Poisson Product

$(a)\ \lambda(t) = \phi(N - i + 1)$ $(b)\ \lambda(t) = \phi(N + i + 1)$

$(c)\ \lambda(t) = \phi(N + i - 1)$ $(d)\ \lambda(t) = \phi(N - i - 1)$

7.26 CMM level 1 has

(a) 6 KPAs (b) 2 KPAs

(c) 0 KPAs (d) None of the above

7.27 MTBF stands for

(a) Mean time between failure (b) Maximum time between failures

(c) Minimum time between failures (d) Many time between failures

7.28 CMM model is a technique to

(a) Improve the software process (b) Automatically develop the software

(c) Test the software (d) All of the above

7.29 Total number of maturing levels in CMM are

(a) 1 (b) 3

(c) 5 (d) 7

Multiple Choice Questions

7.30 Reliability of a software is dependent on number of errors

(a) removed
(b) remaining
(c) both (a) & (b)
(d) None of the above

7.31 Reliability of software is usually estimated at

(a) Analysis phase
(b) Design phase
(c) Coding phase
(d) Testing phase

7.32 CMM stands for

(a) Capacity maturity model
(b) Capability maturity model
(c) Cost management model
(d) Comprehensive maintenance model

7.33 Which level of CMM is for basic project management?

(a) Initial
(b) Repeatable
(c) Defined
(d) Managed

7.34 Which level of CMM is for process management?

(a) Initial
(b) Repeatable
(c) Defined
(d) Optimizing

Multiple Choice Questions

7.35 Which level of CMM is for process management?

(a) Initial
(b) Defined
(c) Managed
(d) Optimizing

7.36 CMM was developed at

(a) Harvard University
(b) Cambridge University
(c) Carnegie Mellon University
(d) Maryland University

7.37 McCall has developed a

(a) Quality model
(b) Process improvement model
(c) Requirement model
(d) Design model

7.38 The model to measure the software process improvement is called

(a) ISO 9000
(b) ISO 9126
(c) CMM
(d) Spiral model

7.39 The number of clauses used in ISO 9001 are

(a) 15
(b) 25
(c) 20
(d) 10

Multiple Choice Questions

7.40 ISO 9126 contains definitions of

(a) quality characteristics
(b) quality factors
(c) quality attributes
(d) All of the above

7.41 In ISO 9126, each characteristics is related to

(a) one attributes
(b) two attributes
(c) three attributes
(d) four attributes

7.42 In McCall quality model; product revision quality factor consist of

(a) Maintainability
(b) Flexibility
(c) Testability
(d) None of the above

7.43 Which is not a software reliability model ?

(a) The Jelinski-Moranda Model
(b) Basic execution time model
(c) Spiral model
(d) None of the above

7.44 Each maturity model is CMM has

(a) One KPA
(b) Equal KPAs
(c) Several KPAs
(d) no KPA

Multiple Choice Questions

7.45 KPA in CMM stands for

(a) Key Process Area
(b) Key Product Area
(c) Key Principal Area
(d) Key Performance Area

7.46 In reliability models, our emphasis is on

(a) errors
(b) faults
(c) failures
(d) bugs

7.47 Software does not break or wear out like hardware. What is your opinion?

(a) True
(b) False
(c) Can not say
(d) not fixed

7.48 Software reliability is defined with respect to

(a) time
(b) speed
(c) quality
(d) None of the above

7.49 MTTF stands for

(a) Mean time to failure
(b) Maximum time to failure
(c) Minimum time to failure
(d) None of the above

Multiple Choice Questions

7.50 ISO 9000 is a series of standards for quality management systems and has

(a) 2 related standards
(b) 5 related standards
(c) 10 related standards
(d) 25 related standards

Exercises

7.1 What is software reliability? Does it exist?

7.2 Explain the significance of bath tube curve of reliability with the help of a diagram.

7.3 Compare hardware reliability with software reliability.

7.4 What is software failure? How is it related with a fault?

7.5 Discuss the various ways of characterising failure occurrences with respect to time.

7.6 Describe the following terms:

(i) Operational profile (ii) Input space

(iii) MTBF (iv) MTTF

(v) Failure intensity.

Exercises

7.7 What are uses of reliability studies? How can one use software reliability measures to monitor the operational performance of software?

7.8 What is software quality? Discuss software quality attributes.

7.9 What do you mean by software quality standards? Illustrate their essence as well as benefits.

7.10 Describe the McCall software quality model. How many product quality factors are defined and why?

7.11 Discuss the relationship between quality factors and quality criteria in McCall's software quality model.

7.12 Explain the Boehm software quality model with the help of a block diagram.

7.13 What is ISO9126 ? What are the quality characteristics and attributes?

Exercises

7.14 Compare the ISO9126 with McCall software quality model and highlight few advantages of ISO9126.

7.15 Discuss the basic model of software reliability. How $\Delta\mu$ *and* $\Delta\tau$ can be calculated.

7.16 Assume that the initial failure intensity is 6 failures/CPU hr. The failure intensity decay parameter is 0.02/failure. We assume that 45 failures have been experienced. Calculate the current failure intensity.

7.17 Explain the basic & logarithmic Poisson model and their significance in reliability studies.

Exercises

7.18 Assume that a program will experience 150 failures in infinite time. It has now experienced 80. The initial failure intensity was 10 failures/CPU hr.

(i) Determine the current failure intensity

(ii) Calculate the failures experienced and failure intensity after 25 and 40 CPU hrs. of execution.

(iii) Compute additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.

Use the basic execution time model for the above mentioned calculations.

7.19 Write a short note on Logarithmic Poisson Execution time model. How can we calculate $\Delta\mu$ & $\Delta\tau$?

7.20 Assume that the initial failure intensity is 10 failures/CPU hr. The failure intensity decay parameter is 0.03/failure. We have experienced 75 failures upto this time. Find the failures experienced and failure intensity after 25 and 50 CPU hrs. of execution.

Exercises

7.21 The following parameters for basic and logarithmic Poisson models are given:

Basic execution time model	*Logarithmic Poisson execution time model*
λ_0 = 5 failures/CPU hr	λ_0 = 25 failures/CPU hr
V_0 = 125 failures	θ = 0.3/failure

Determine the additional failures and additional execution time required to reach the failure intensity objective of 0.1 failure/CPU hr. for both models.

7.22 Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss them.

7.23 Discuss the calendar time component model. Establish the relationship between calendar time to execution time.

Exercises

7.24 A program is expected to have 250 faults. It is also assumed that one fault may lead to one failure. The initial failure intensity is 5 failure/CPU hr. The program is released with a failure intensity objective of 4 failures/10 CPU hr. Calculate the number of failures experienced before release.

7.25 Explain the Jelinski-Moranda model of reliability theory. What is the relation between 't' and $'\lambda'$?

7.26 Describe the Mill's bug seeding model. Discuss few advantages of this model over other reliability models.

7.27 Explain how the CMM encourages continuous improvement of the software process.

7.28 Discuss various key process areas of CMM at various maturity levels.

7.29 Construct a table that correlates key process areas (KPAs) in the CMM with ISO9000.

7.30 Discuss the 20 clauses of ISO9001 and compare with the practices in the CMM.

Exercises

7.31 List the difference of CMM and ISO9001. Why is it suggested that CMM is the better choice than ISO9001?

7.32 Explain the significance of software reliability engineering. Discuss the advantage of using any software standard for software development?

7.33 What are the various key process areas at defined level in CMM? Describe activities associated with one key process area.

7.34 Discuss main requirements of ISO9001 and compare it with SEI capability maturity model.

7.35 Discuss the relative merits of ISO9001 certification and the SEI CMM based evaluation. Point out some of the shortcomings of the ISO9001 certification process as applied to the software industry.

Software Testing

Software Testing

- What is Testing?

Many people understand many definitions of testing **:**

1. Testing is the process of demonstrating that errors are not present.
2. The purpose of testing is to show that a program performs its intended functions correctly.
3. Testing is the process of establishing confidence that a program does what it is supposed to do.

These definitions are incorrect.

Software Testing

A more appropriate definition is:

"*Testing is the process of executing a program with the intent of finding errors*."

Software Testing

- Why should We Test ?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

Software Testing

- Who should Do the Testing ?
  - Testing requires the developers to find errors from their software.
  - It is difficult for software developer to point out errors from own creations.
  - Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

Software Testing

- What should We Test ?

We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are $2^8 x 2^8$. If only one second it required to execute one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.

Some Terminologies

- **Error, Mistake, Bug, Fault and Failure**

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "**bugs**".

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

➢ Test, Test Case and Test Suite

Test and **Test case** terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig. 2: Test case template

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

Software Testing

- Verification and Validation

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

Testing= Verification+Validation

Software Testing

➢ Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Functional Testing

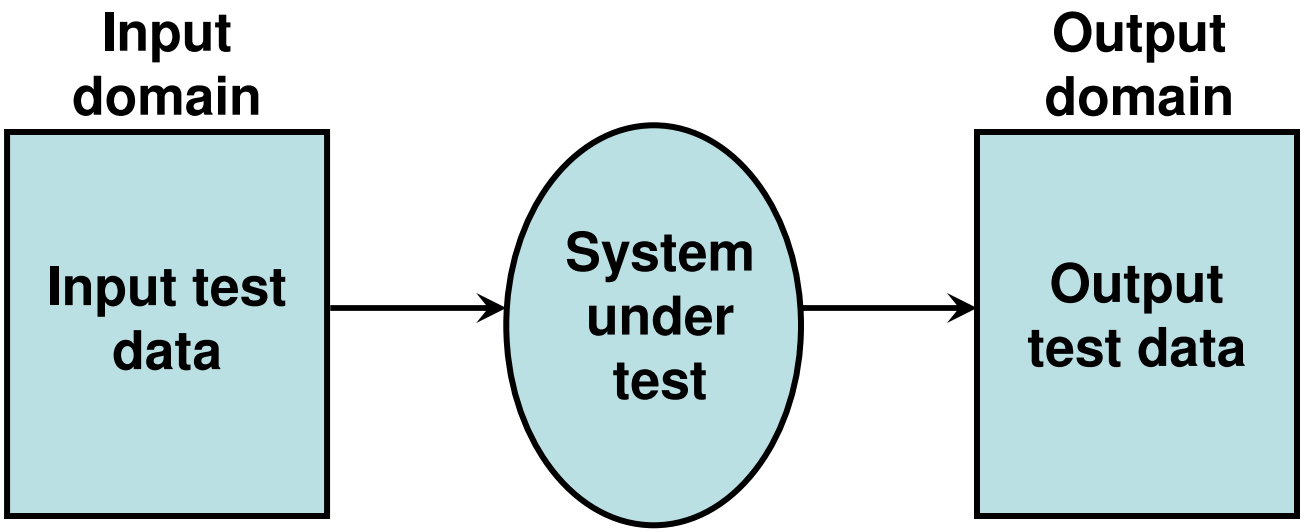


Fig. 3: Black box testing

Boundary Value Analysis

Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \leq x \leq b$$
$$c \leq y \leq d$$

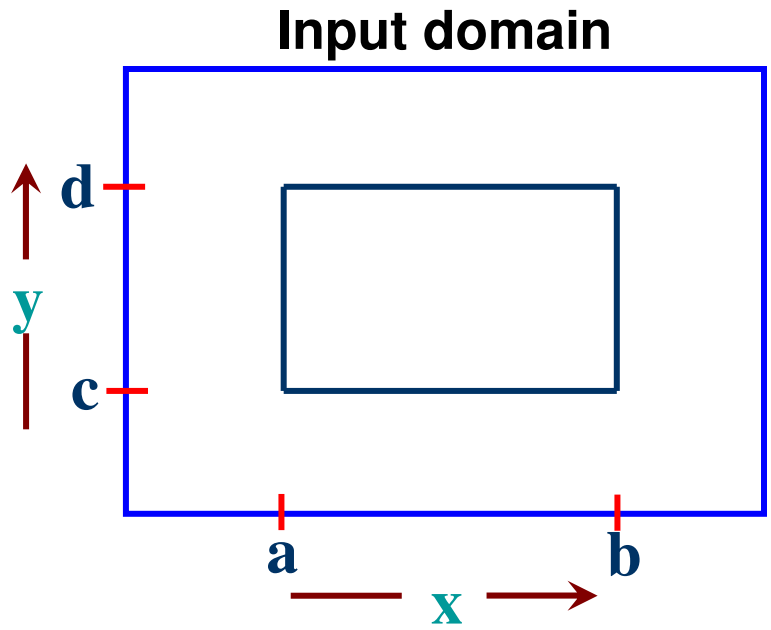


Fig.4: Input domain for program having two input variables

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). This input domain is shown in Fig. 8.5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yield **4n + 1** test cases.

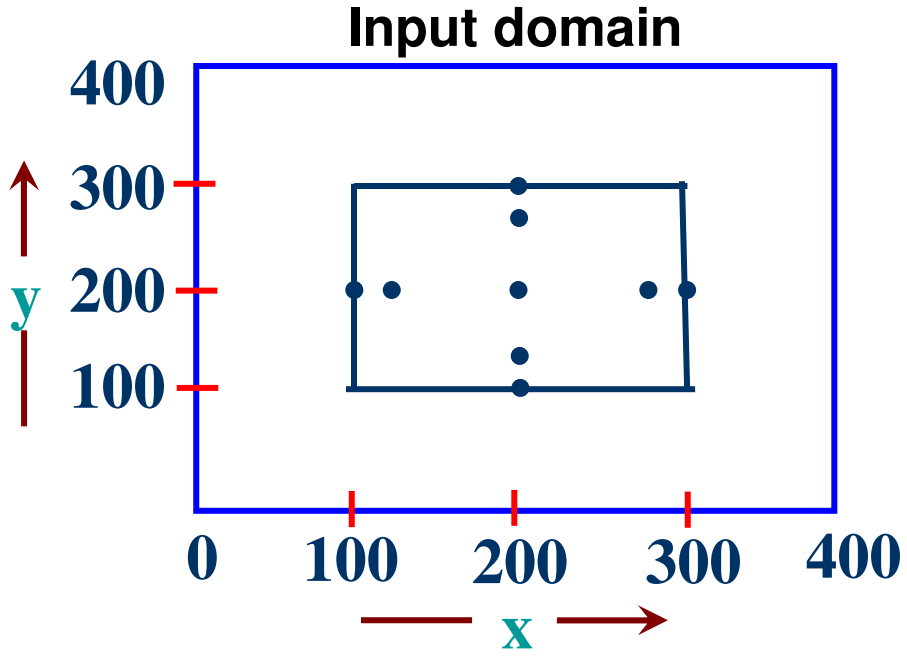


Fig. 5: Input domain of two variables x and y with boundaries [100,300] each

Example- 8.I

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

The boundary value test cases are :

Test Case	a	b	c	Expected output
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Example – 8.2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Solution

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.

Software Testing

The boundary value test cases are:

Test Case	Month	Day	Year	Expected output
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Example – 8.3

Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle]

Design the boundary value test cases.

Solution

The boundary value test cases are shown below:

Test case	x	y	z	Expected Output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle

Robustness testing

It is nothing but the extension of boundary value analysis. Here, we would like to see, what happens when the extreme values are exceeded with a value slightly greater than the maximum, and a value slightly less than minimum. It means, we want to go outside the legitimate boundary of input domain. This extended form of boundary value analysis is called robustness testing and shown in Fig. 6

There are four additional test cases which are outside the legitimate input domain. Hence total test cases in robustness testing are 6n+1, where n is the number of input variables. So, 13 test cases are:

(200,99), (200,100), (200,101), (200,200), (200,299), (200,300)

(200,301), (99,200), (100,200), (101,200), (299,200), (300,200), (301,200)

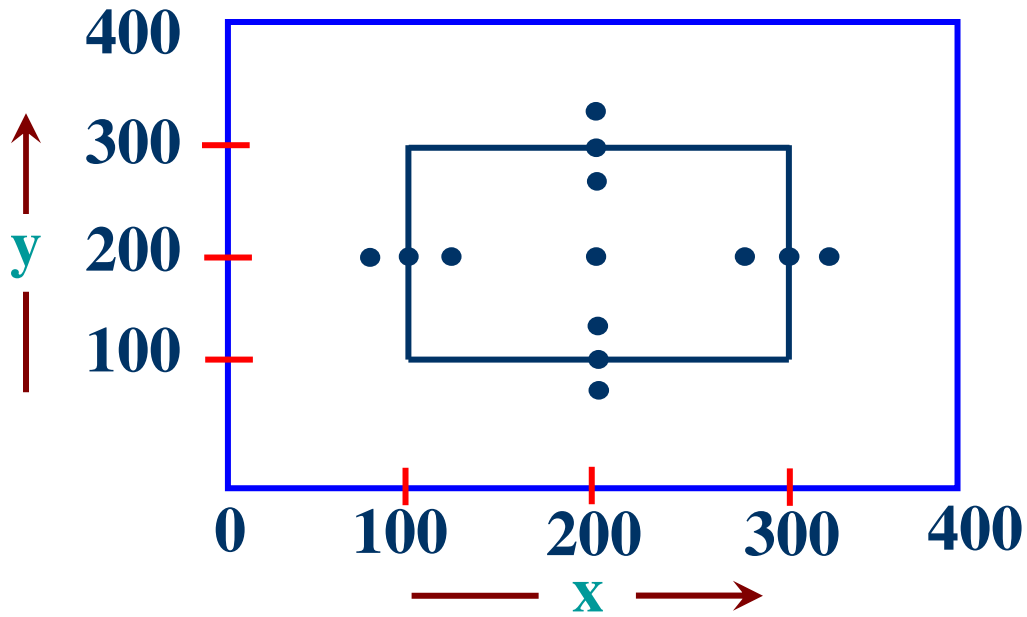


Fig. 8.6: Robustness test cases for two variables x and y with range [100,300] each

Example - 8.4

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Design the Robust test case and worst test cases for this program.

Solution

Robust test cases are 6n+1. Hence, in 3 variable input cases total number of test cases are 19 as given on next slide:

Test case	a	b	c	Expected Output
1	-1	50	50	Invalid input`
2	0	50	50	Not quadratic equation
3	1	50	50	Real roots
4	50	50	50	Imaginary roots
5	99	50	50	Imaginary roots
6	100	50	50	Imaginary roots
7	101	50	50	Invalid input
8	50	-1	50	Invalid input
9	50	0	50	Imaginary roots
10	50	1	50	Imaginary roots
11	50	99	50	Imaginary roots
12	50	100	50	Equal roots
13	50	101	50	Invalid input
14	50	50	-1	Invalid input
15	50	50	0	Real roots
16	50	50	1	Real roots
17	50	50	99	Imaginary roots
18	50	50	100	Imaginary roots
19	50	50	101	Invalid input

Example – 8.5

Consider the program for the determination of previous date in a calendar as explained in example 8.2. Design the robust and worst test cases for this program.

Solution

Robust test cases are 6*n*+1. Hence total 19 robust test cases are designed and are given on next slide.

Test case	Month	Day	Year	Expected Output
1	6	15	1899	Invalid date (outside range)
2	6	15	1900	14 June, 1900
3	6	15	1901	14 June, 1901
4	6	15	1962	14 June, 1962
5	6	15	2024	14 June, 2024
6	6	15	2025	14 June, 2025
7	6	15	2026	Invalid date (outside range)
8	6	0	1962	Invalid date
9	6	1	1962	31 May, 1962
10	6	2	1962	1 June, 1962
11	6	30	1962	29 June, 1962
12	6	31	1962	Invalid date
13	6	32	1962	Invalid date
14	0	15	1962	Invalid date
15	1	15	1962	14 January, 1962
16	2	15	1962	14 February, 1962
17	11	15	1962	14 November, 1962
18	12	15	1962	14 December, 1962
19	13	15	1962	Invalid date

Example – 8.6

Consider the triangle problem as given in example 8.3. Generate robust and worst test cases for this problem.

Solution

Robust test cases are given on next slide.

`	x	y	z	Expected Output
1	50	50	0	Invalid input`
2	50	50	1	Isosceles
3	50	50	2	Isosceles
4	50	50	50	Equilateral
5	50	50	99	Isosceles
6	50	50	100	Not a triangle
7	50	50	101	Invalid input
8	50	0	50	Invalid input
9	50	1	50	Isosceles
10	50	2	50	Isosceles
11	50	99	50	Isosceles
12	50	100	50	Not a triangle
13	50	101	50	Invalid input
14	0	50	50	Invalid input
15	1	50	50	Isosceles
16	2	50	50	Isosceles
17	99	50	50	Isosceles
18	100	50	50	Not a triangle
19	100	50	50	Invalid input

Structural Testing

A complementary approach to functional testing is called structural / white box testing. It permits us to examine the internal structure of the program.

Path Testing

Path testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that we have achieved some measure of test thoroughness.

This type of testing involves:

1. generating a set of paths that will cover every branch in the program.
2. finding a set of test cases that will execute every path in the set of program paths.

Flow Graph

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control.

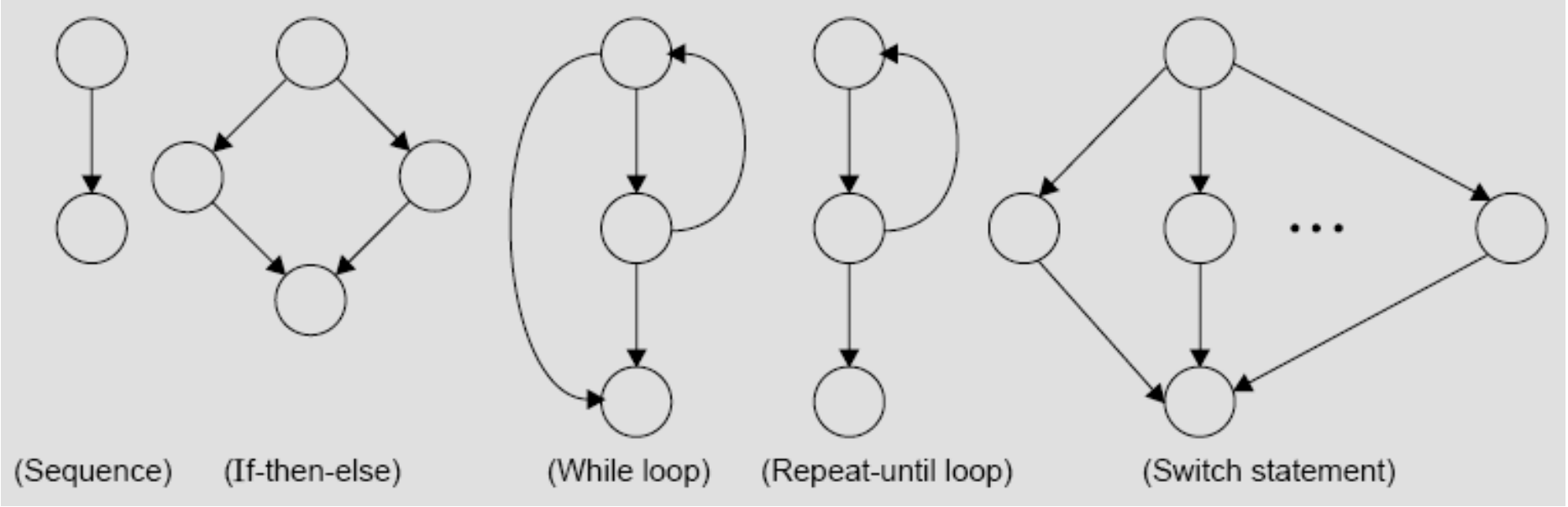


Fig. 14: The basic construct of the flow graph

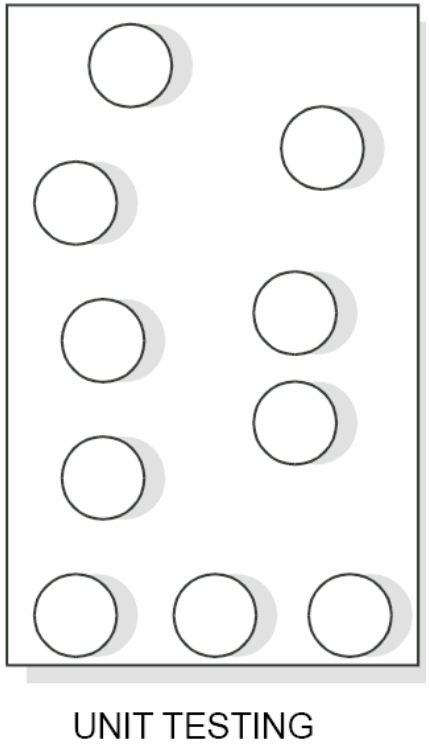
Levels of Testing

There are 3 levels of testing:

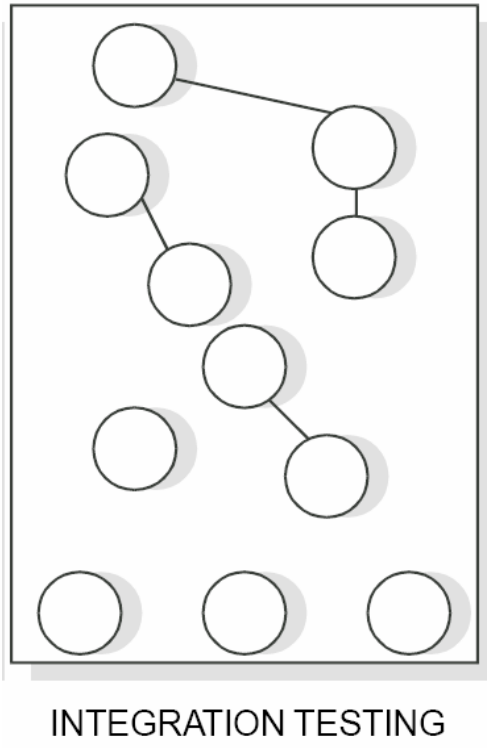
i. Unit Testing

ii. Integration Testing

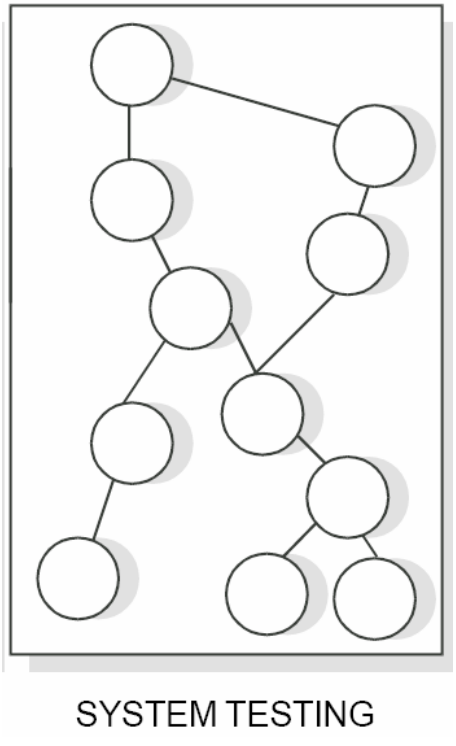
iii. System Testing



UNIT TESTING



INTEGRATION TESTING



SYSTEM TESTING

Unit Testing

There are number of reasons in support of unit testing than testing the entire product.

1. The size of a single module is small enough that we can locate an error fairly easily.

2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.

3. Confusing interactions of multiple errors in widely different parts of the software are eliminated.

Software Testing

There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? One approach is to construct an appropriate driver routine to call if and, simple stubs to be called by it, and to insert output statements in it.

Stubs serve to replace modules that are subordinate to (called by) the module to be tested. A stub or dummy subprogram uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns.

This overhead code, called scaffolding represents effort that is import to testing, but does not appear in the delivered product as shown in Fig. 29.

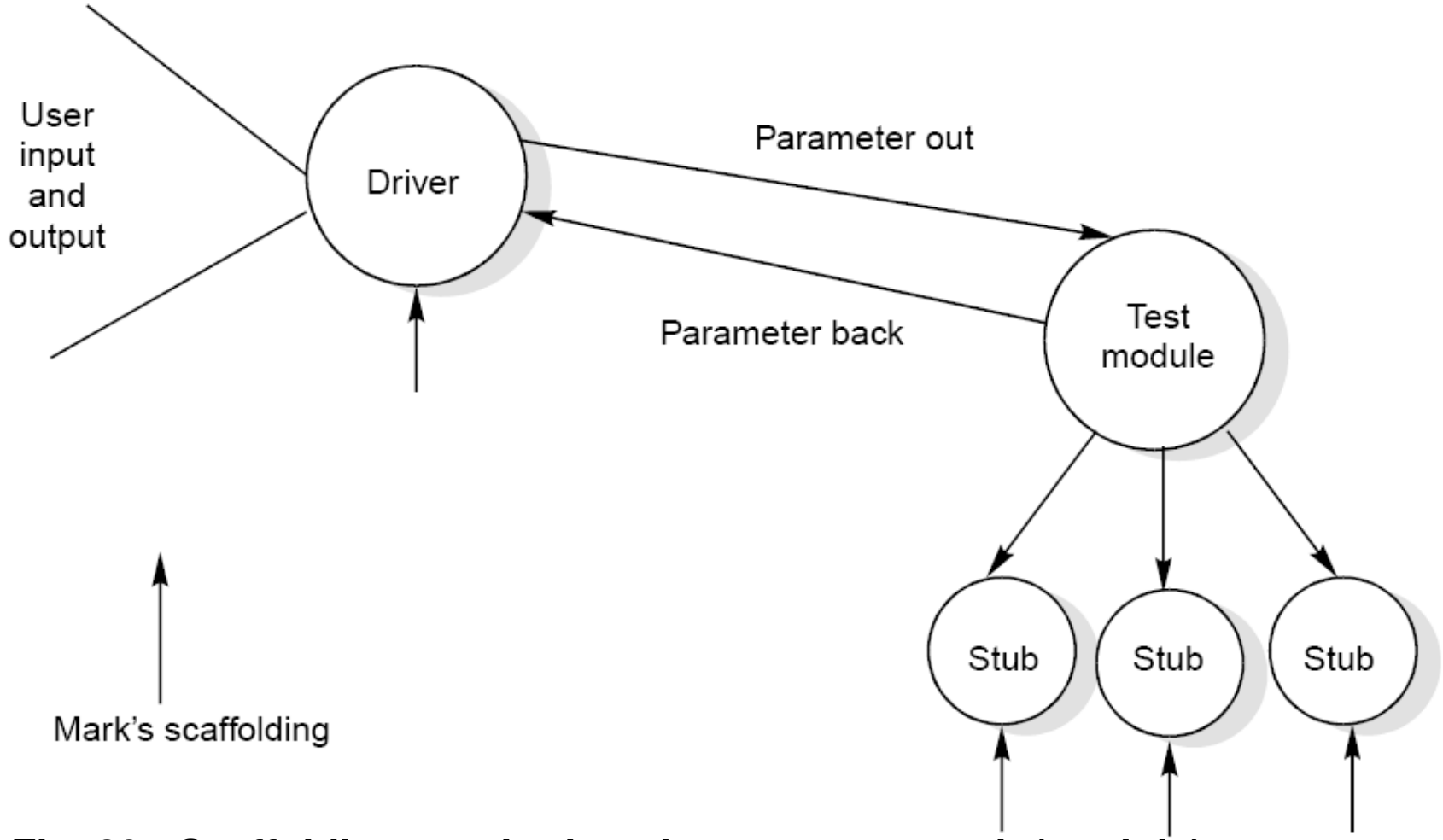


Fig. 29 : Scaffolding required testing a program unit (module)

Integration Testing

The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed. One specific target of integration testing is the interface: whether parameters match on both sides as to type, permissible ranges, meaning and utilization.

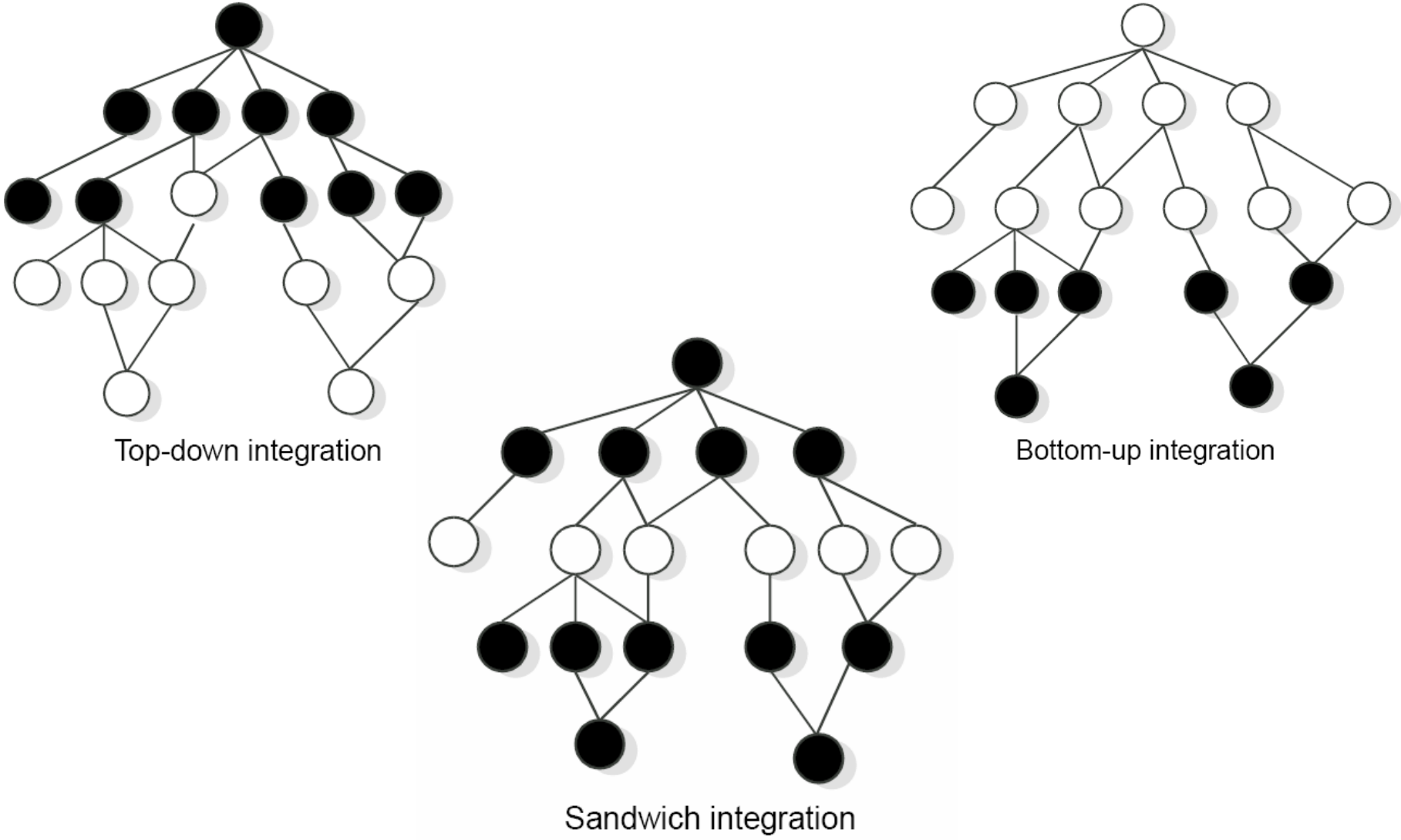


Fig. 30 : Three different integration approaches

System Testing

Of the three levels of testing, the system level is closet to everyday experiences. We test many things; a used car before we buy it, an on-line cable network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations; not with respect to a specification or a standard. Consequently, goal is not to find faults, but to demonstrate performance. Because of this we tend to approach system testing from a functional standpoint rather than from a structural one. Since it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and is compounded by the reduced testing interval that usually remains before a delivery deadline.

Petschenik gives some guidelines for choosing test cases during system testing.

Software Testing

During system testing, we should evaluate a number of attributes of the software that are vital to the user and are listed in Fig. 31. These represent the operational correctness of the product and may be part of the software specifications.

Usable	Is the product convenient, clear, and predictable?
Secure	Is access to sensitive data restricted to those with authorization?
Compatible	Will the product work correctly in conjunction with existing data, software, and procedures?
Dependable	Do adequate safeguards against failure and methods for recovery exist in the product?
Documented	Are manuals complete, correct, and understandable?

Fig. 31 : Attributes of software to be tested during system testing

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

8.1 Software testing is:

(a) the process of demonstrating that errors are not present

(b) the process of establishing confidence that a program does what it is supposed to do

(c) the process of executing a program to show it is working as per specifications

(d) the process of executing a program with the intent of finding errors

8.2 Software mistakes during coding are known as:

(a) failures (b) defects

(c) bugs (d) errors

8.3 Functional testing is known as:

(a) Structural testing (b) Behavior testing

(c) Regression testing (d) None of the above

8.4 For a function of n variables, boundary value analysis yields:

(a) 4n+3 test cases (b) 4n+1 test cases

(c) n+4 test cases (d) None of the above

Multiple Choice Questions

8.5 For a function of two variables, how many cases will be generated by robustness testing?

(a) 9 (b) 13

(c) 25 (d) 42

8.6 For a function of n variables robustness testing of boundary value analysis yields:

(a) 4n+1 (b) 4n+3

(c) 6n+1 (d) None of the above

8.7 Regression testing is primarily related to:

(a) Functional testing (b) Data flow testing

(c) Development testing (d) Maintenance testing

8.8 A node with indegree=0 and out degree $\neq$ 0 is called

(a) Source node (b) Destination node

(c) Transfer node (d) None of the above

8.9 A node with indegree $\neq$ 0 and out degree=0 is called

(a) Source node (b) Predicate node

(c) Destination node (d) None of the above

Multiple Choice Questions

8.10 A decision table has

(a) Four portions (b) Three portions

(c) Five portions (d) Two portions

8.11 Beta testing is carried out by

(a) Users (b) Developers

(c) Testers (d) All of the above

8.12 Equivalence class partitioning is related to

(a) Structural testing (b) Blackbox testing

(c) Mutation testing (d) All of the above

8.13 Cause effect graphing techniques is one form of

(a) Maintenance testing (b) Structural testing

(c) Function testing (d) Regression testing

8.14 During validation

(a) Process is checked (b) Product is checked

(c) Developer's performance is evaluated (d) The customer checks the product

Multiple Choice Questions

8.15 Verification is

(a) Checking the product with respect to customer's expectation

(b) Checking the product with respect to specifications

(c) Checking the product with respect to the constraints of the project

(d) All of the above

8.16 Validation is

(a) Checking the product with respect to customer's expectation

(b) Checking the product with respect to specifications

(c) Checking the product with respect to the constraints of the project

(d) All of the above

8.17 Alpha testing is done by

(a) Customer

(b) Tester

(c) Developer

(d) All of the above

8.18 Site for Alpha testing is

(a) Software company

(b) Installation place

(c) Any where

(d) None of the above

Multiple Choice Questions

8.19 Site for Beta testing is

(a) Software company
(b) User's site
(c) Any where
(d) All of the above

8.20 Acceptance testing is done by

(a) Developers
(b) Customers
(c) Testers
(d) All of the above

8.21 One fault may lead to

(a) One failure
(b) No failure
(c) Many failure
(d) All of the above

8.22 Test suite is

(a) Set of test cases
(b) Set of inputs
(c) Set of outputs
(d) None of the above

8.23 Behavioral specification are required for:

(a) Modeling
(b) Verification
(c) Validation
(d) None of the above

Multiple Choice Questions

8.24 During the development phase, the following testing approach is not adopted

(a) Unit testing
(b) Bottom up testing
(c) Integration testing
(d) Acceptance testing

8.25 Which is not a functional testing technique?

(a) Boundary value analysis
(b) Decision table
(c) Regression testing
(d) None of the above

8.26 Decision table are useful for describing situations in which:

(a) An action is taken under varying sets of conditions.
(b) Number of combinations of actions are taken under varying sets of conditions
(c) No action is taken under varying sets of conditions
(d) None of the above

8.27 One weakness of boundary value analysis and equivalence partitioning is

(a) They are not effective
(b) They do not explore combinations of input circumstances
(c) They explore combinations of input circumstances
(d) None of the above

Multiple Choice Questions

8.28 In cause effect graphing technique, cause & effect are related to

(a) Input and output
(b) Output and input
(c) Destination and source
(d) None of the above

8.29 DD path graph is called as

(a) Design to Design Path graph
(b) Defect to Defect Path graph
(c) Destination to Destination Path graph
(d) Decision to decision Path graph

8.30 An independent path is

(a) Any path through the DD path graph that introduce at least one new set of processing statements or new conditions

(b) Any path through the DD path graph that introduce at most one new set of processing statements or new conditions

(c) Any path through the DD path graph that introduce at one and only one new set of processing statements or new conditions

(d) None of the above

8.31 Cyclomatic complexity is developed by

(a) B.W.Boehm
(b) T.J.McCabe
(c) B.W.Lettlewood
(d) Victor Basili

Multiple Choice Questions

8.32 Cyclomatic complexity is denoted by

(a) V(G)=e-n+2P (b) V(G)= ∏ +1

(c) V(G)=Number of regions of the graph (d) All of the above

8.33 The equation V(G)= ∏ +1 of cyclomatic complexity is applicable only if every predicate node has

(a) two outgoing edges (b) three or more outgoing edges

(c) no outgoing edges (d) none of the above

8.34 The size of the graph matrix is

(a) Number of edges in the flow graph

(b) Number of nodes in the flow graph

(c) Number of paths in the flow graph

(d) Number of independent paths in the flow graph

Multiple Choice Questions

8.35 Every node is represented by

(a) One row and one column in graph matrix

(b) Two rows and two columns in graph matrix

(c) One row and two columns in graph matrix

(d) None of the above

8.36 Cyclomatic complexity is equal to

(a) Number of independent paths (b) Number of paths

(c) Number of edges (d) None of the above

8.37 Data flow testing is related to

(a) Data flow diagrams (b) E-R diagrams

(c) Data dictionaries (d) none of the above

8.38 In data flow testing, objective is to find

(a) All dc-paths that are not du-paths (b) All du-paths

(c) All du-paths that are not dc-paths (d) All dc-paths

Multiple Choice Questions

8.39 Mutation testing is related to

(a) Fault seeding
(b) Functional testing
(c) Fault checking
(d) None of the above

8.40 The overhead code required to be written for unit testing is called

(a) Drivers
(b) Stubs
(c) Scaffolding
(d) None of the above

8.41 Which is not a debugging techniques

(a) Core dumps
(b) Traces
(c) Print statements
(d) Regression testing

8.42 A break in the working of a system is called

(a) Defect
(b) Failure
(c) Fault
(d) Error

8.43 Alpha and Beta testing techniques are related to

(a) System testing
(b) Unit testing
(c) acceptance testing
(d) Integration testing

Multiple Choice Questions

8.44 Which one is not the verification activity

(a) Reviews
(b) Path testing
(c) Walkthrough
(d) Acceptance testing

8.45 Testing the software is basically

(a) Verification
(b) Validation
(c) Verification and validation
(d) None of the above

8.46 Integration testing techniques are

(a) Topdown
(b) Bottom up
(c) Sandwich
(d) All of the above

8.47 Functionality of a software is tested by

(a) White box testing
(b) Black box testing
(c) Regression testing
(d) None of the above

8.48 Top down approach is used for

(a) Development
(b) Identification of faults
(c) Validation
(d) Functional testing

Multiple Choice Questions

8.49 Thread testing is used for testing

(a) Real time systems
(b) Object oriented systems
(c) Event driven systems
(d) All of the above

8.50 Testing of software with actual data and in the actual environment is called

(a) Alpha testing
(b) Beta testing
(c) Regression testing
(d) None of the above

Exercises

8.1 What is software testing? Discuss the role of software testing during software life cycle and why is it so difficult?

8.2 Why should we test? Who should do the testing?

8.3 What should we test? Comment on this statement. Illustrate the importance of testing

8.4 Defined the following terms:

(i) fault (ii) failure

(iii) bug (iv) mistake

8.5 What is the difference between

(i) Alpha testing & beta testing

(ii) Development & regression testing

(iii) Functional & structural testing

8.6 Discuss the limitation of testing. Why do we say that complete testing is impossible?

Exercises

8.7 Briefly discuss the following

(i) Test case design, Test & Test suite

(ii) Verification & Validation

(iii) Alpha, beta & acceptance testing

8.8 Will exhaustive testing (even if possible for every small programs) guarantee that the program is 100% correct?

8.9 Why does software fail after it has passed from acceptance testing? Explain.

8.10 What are various kinds of functional testing? Describe any one in detail.

8.11 What is a software failure? Explain necessary and sufficient conditions for software failure. Mere presence of faults means software failure. Is it true? If not, explain through an example, a situation in which a failure will definitely occur.

8.12 Explain the boundary value analysis testing techniques with the help of an example.

Exercises

8.13 Consider the program for the determination of next date in a calendar. Its input is a triple of day, month and year with the following range

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1900 \quad 1 \leq \text{year} \leq 2025$

The possible outputs would be Next date or invalid date. Design boundary value, robust and worst test cases for this programs.

8.14 Discuss the difference between worst test case and adhoc test case performance evaluation by means of testing. How can we be sure that the real worst case has actually been observed?

8.15 Describe the equivalence class testing method. Compare this with boundary value analysis techniques

Exercises

8.16 Consider a program given below for the selection of the largest of numbers

```
main()
    {
      float A,B,C;
      printf("Enter three values\n");
      scanf("%f%f%f",  &A,&B,&c);
      printf("\n Largest value is");
      if (A>B)
           {
                if(A>C)
                     printf("%f\n",A);
                else
                     printf("%f\n",c);
           }
           else
           {
                if(C>B)
                     printf("%f\n",C);
                else
                     printf("%f\n",B);
           }
    }
```

Exercises

(i) Design the set of test cases using boundary value analysis technique and equivalence class testing technique.

(ii) Select a set of test cases that will provide 100% statement coverage.

(iii) Develop a decision table for this program.

8.17 Consider a small program and show, why is it practically impossible to do exhaustive testing?

8.18 Explain the usefulness of decision table during testing. Is it really effective? Justify your answer.

8.19 Draw the cause effect graph of the program given in exercise 8.16.

8.20 Discuss cause effect graphing technique with an example.

8.21 Determine the boundary value test cases the extended triangle problem that also considers right angle triangles.

Exercises

8.22 Why does software testing need extensive planning? Explain.

8.23 What is meant by test case design? Discuss its objectives and indicate the steps involved in test case design.

8.24 Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Marks obtained	*Grade*
80–100	Distinction
60–79	First division
50–59	Second division
40–49	Third division
0–39	Fail

Generate test cases using equivalence class testing technique

Exercises

8.25 Consider a program to determine whether a number is 'odd' or 'even' and print the message

NUMBER IS EVEN

Or

NUMBER IS ODD

The number may be any valid integer.

Design boundary value and equivalence class test cases.

8.26 Admission to a professional course is subject to the following conditions:

(*a*) Marks in Mathematics >=	60
(*b*) Marks in Physics >=	50
(*c*) Marks in Chemistry >=	40
(*d*) Total in all three subjects >=	200
Or	
Total in Mathematics and Physics >=	150

Exercises

If aggregate marks of an eligible candidate are more than 225, he/she will be eligible for honors course, otherwise he/she will be eligible for pass course. The program reads the marks in the three subjects and generates the following outputs:

(a) Not Eligible

(b) Eligible to Pass Course

(c) Eligible to Honors Course

Design test cases using decision table testing technique.

8.27 Draw the flow graph for program of largest of three numbers as shown in exercise 8.16. Find out all independent paths that will guarantee that all statements in the program have been tested.

8.28 Explain the significance of independent paths. Is it necessary to look for a tool for flow graph generation, if program size increases beyond 100 source lines?

8.29 Discuss the structure testing. How is it different form functional testing?

Exercises

8.30 What do you understand by structural testing? Illustrate important structural testing techniques.

8.31 Discuss the importance of path testing during structural testing.

8.32 What is cyclomatic complexity? Explain with the help of an example.

8.33 Is it reasonable to define "thresholds" for software modules? For example, is a module acceptable if its $V(G) \leq 10$? Justify your answer.

8.34 Explain data flow testing. Consider an example and show all "du" paths. Also identify those "du" paths that are not "dc" paths.

8.35 Discuss the various steps of data flow testing.

8.36 If we perturb a value, changing the current value of 100 by 1000, what is the effect of this change? What precautions are required while designing the test cases?

Exercises

8.37 What is the difference between white and black box testing? Is determining test cases easier in back or white box testing? Is it correct to claim that if white box testing is done properly, it will achieve close to 100% path coverage?

8.38 What are the objectives of testing? Why is the psychology of a testing person important?

8.39 Why does software fail after it has passed all testing phases? Remember, software, unlike hardware does not wear out with time.

8.40 What is the purpose of integration testing? How is it done?

8.41 Differentiate between integration testing and system testing.

8.42 Is unit testing possible or even desirable in all circumstances? Provide examples to Justify your answer?

8.43 Peteschenik suggested that a different team than the one that does integration testing should carry out system testing. What are some good reasons for this?

Exercises

8.44 Test a program of your choice, and uncover several program errors. Localise the main route of these errors, and explain how you found the courses. Did you use the techniques of Table 8? Explain why or why not.

8.45 How can design attributes facilitate debugging?

8.46 List some of the problem that could result from adding debugging statements to code. Discuss possible solutions to these problems.

8.47 What are various debugging approaches? Discuss them with the help of examples.

8.48 Researchers and practitioners have proposed several mixed testing strategies intended to combine advantages of various techniques discussed in this chapter. Propose your own combination, perhaps also using some kind of random testing at selected points.

8.49 Design a test set for a spell checker. Then run it on a word processor having a spell checker, and report on possible inadequacies with respect to your requirements.

Exercises

8.50 4 GLs represent a major step forward in the development of automatic program generation. Explain the major advantage & disadvantage in the use of 4 GLs. What are the cost impact of applications of testing and how do you justify expenditures for these activities.

Software Maintenance

Software Maintenance

What is Software Maintenance?

Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization.

Software Maintenance

Categories of Maintenance

- Corrective maintenance

This refer to modifications initiated by defects in the software.

- Adaptive maintenance

It includes modifying the software to match changes in the ever changing environment.

- Perfective maintenance

It means improving processing efficiency or performance, or restructuring the software to improve changeability. This may include enhancement of existing system functionality, improvement in computational efficiency etc.

Software Maintenance

- Other types of maintenance

There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflect deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible. This work may be named as preventive maintenance.

Software Maintenance

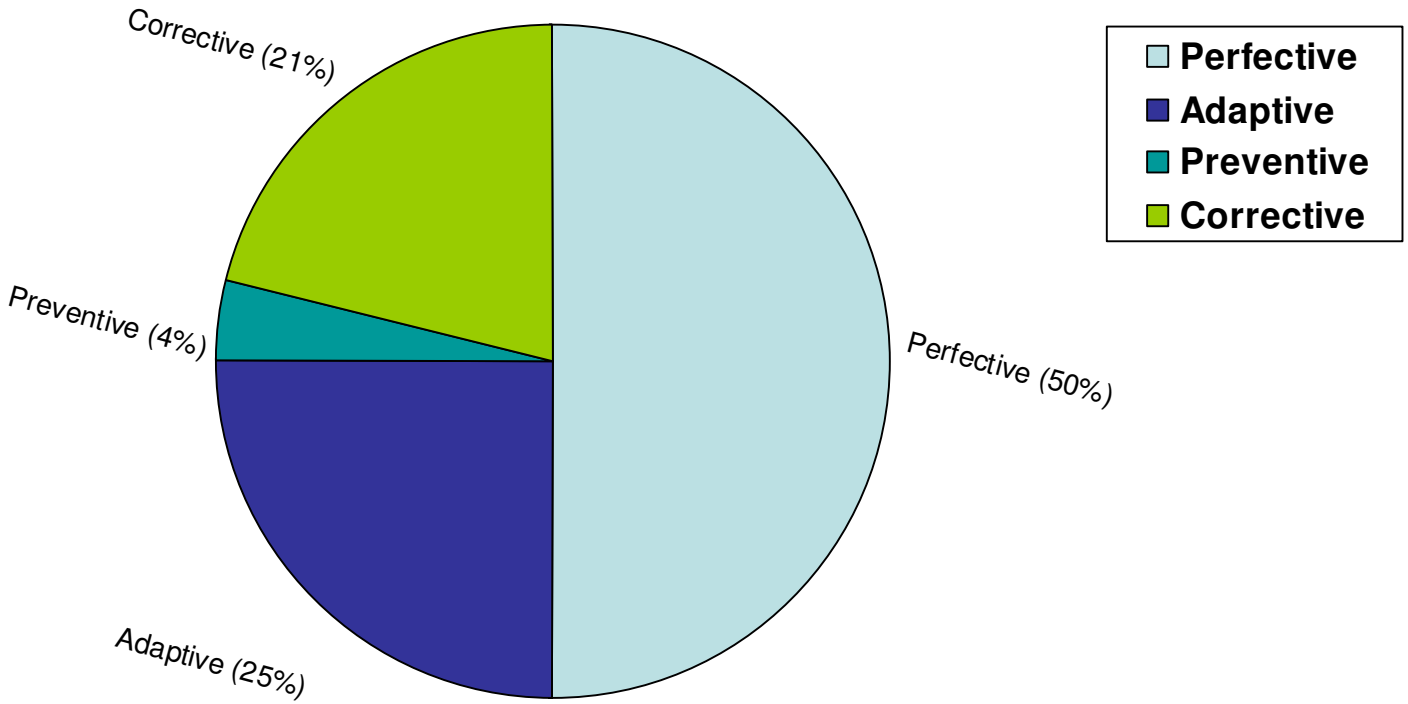


Fig. 1: Distribution of maintenance effort

Software Maintenance

Problems During Maintenance

- Often the program is written by another person or group of persons.
- Often the program is changed by person who did not understand it clearly.
- Program listings are not structured.
- High staff turnover.
- Information gap.
- Systems are not designed for change.

Maintenance is Manageable

A common misconception about maintenance is that it is not manageable.

Report of survey conducted by Lientz & Swanson gives some interesting observations:

1	Emergency debugging	12.4%
2	Routine debugging	9.3%
3	Data environment adaptation	17.3%
4	Changes in hardware and OS	6.2%
5	Enhancements for users	41.8%
6	Documentation Improvement	5.5%
7	Code efficiency improvement	4.0%
8	Others	3.5%

Table 1: Distribution of maintenance effort

Software Maintenance

Kinds of maintenance requests

1	New reports	40.8%
2	Add data in existing reports	27.1%
3	Reformed reports	10%
4	Condense reports	5.6%
5	Consolidate reports	6.4%
6	Others	10.1%

Table 2: Kinds of maintenance requests

Software Maintenance

Potential Solutions to Maintenance Problems

- Budget and effort reallocation
- Complete replacement of the system
- Maintenance of existing system

Software Maintenance

The Maintenance Process

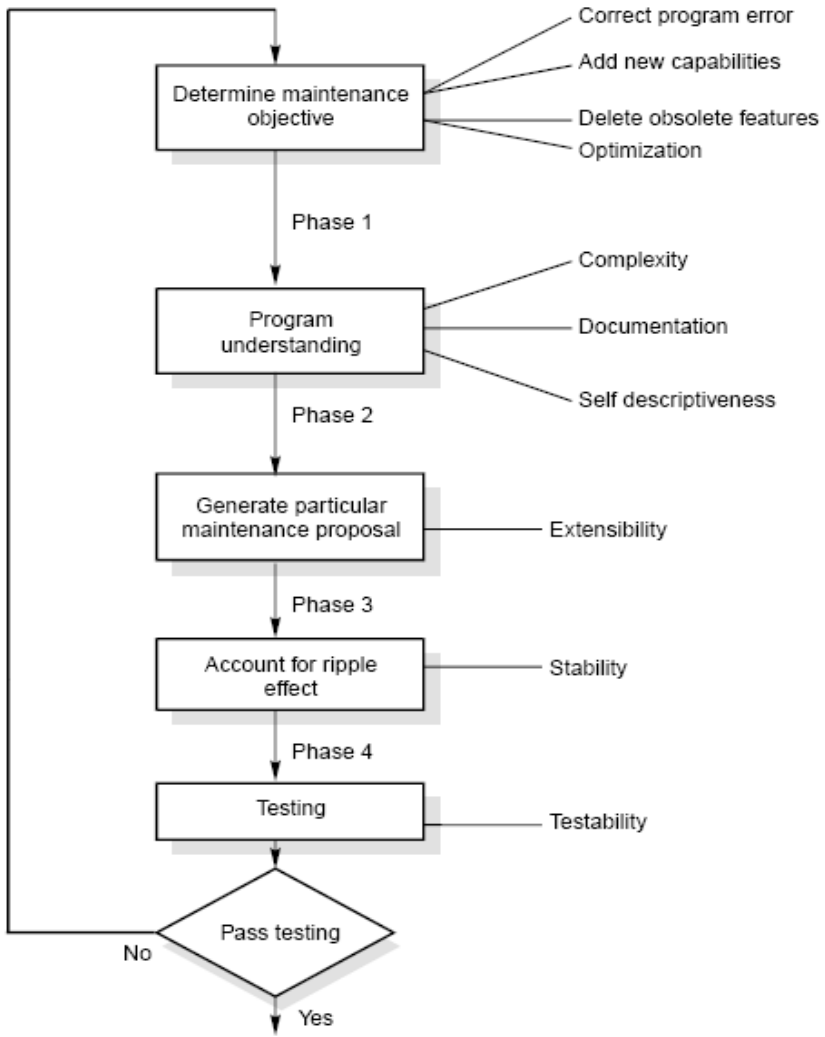


Fig. 2: The software maintenance process

Software Maintenance

- Program Understanding

The first phase consists of analyzing the program in order to understand.

- Generating Particular Maintenance Proposal

The second phase consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective.

- Ripple Effect

The third phase consists of accounting for all of the ripple effect as a consequence of program modifications.

Software Maintenance

- Modified Program Testing

The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before.

- Maintainability

Each of these four phases and their associated software quality attributes are critical to the maintenance process. All of these factors must be combined to form maintainability.

Maintenance Models

- Quick-fix Model

This is basically an adhoc approach to maintaining software. It is a fire fighting approach, waiting for the problem to occur and then trying to fix it as quickly as possible.

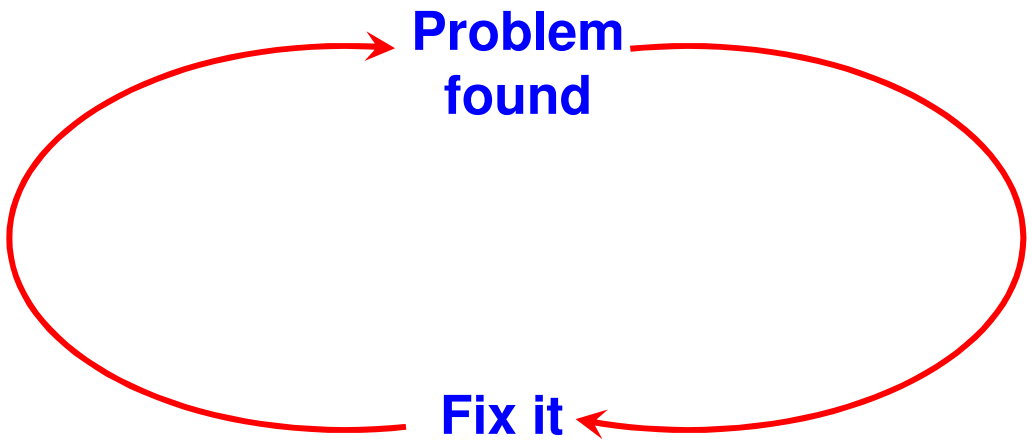


Fig. 3: The quick-fix model

Software Maintenance

- Iterative Enhancement Model
  - Analysis
  - Characterization of proposed modifications
  - Redesign and implementation

Software Maintenance

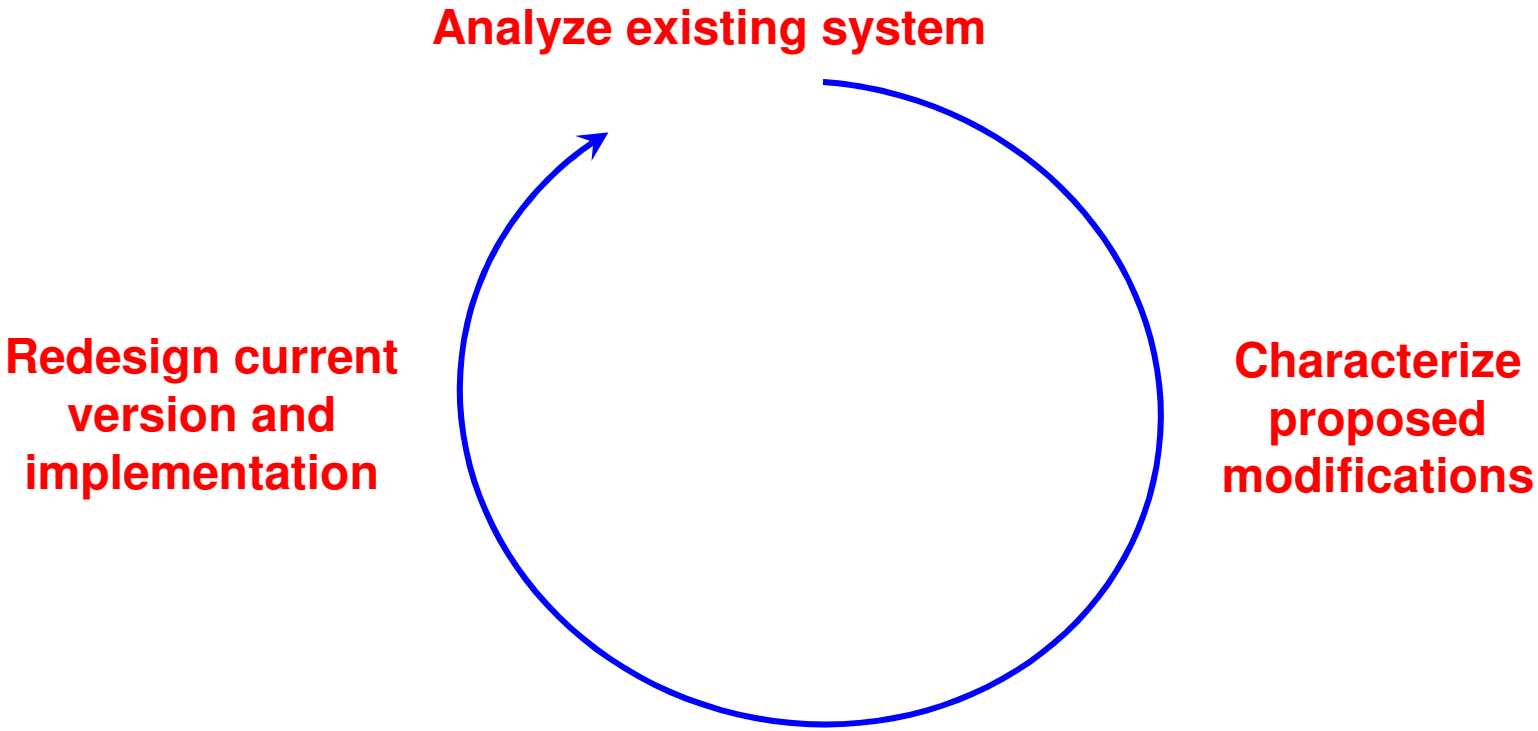


Fig. 4: The three stage cycle of iterative enhancement

Software Maintenance

- Reuse Oriented Model

The reuse model has four main steps:

1. Identification of the parts of the old system that are candidates for reuse.
2. Understanding these system parts.
3. Modification of the old system parts appropriate to the new requirements.
4. Integration of the modified parts into the new system.

Software Maintenance

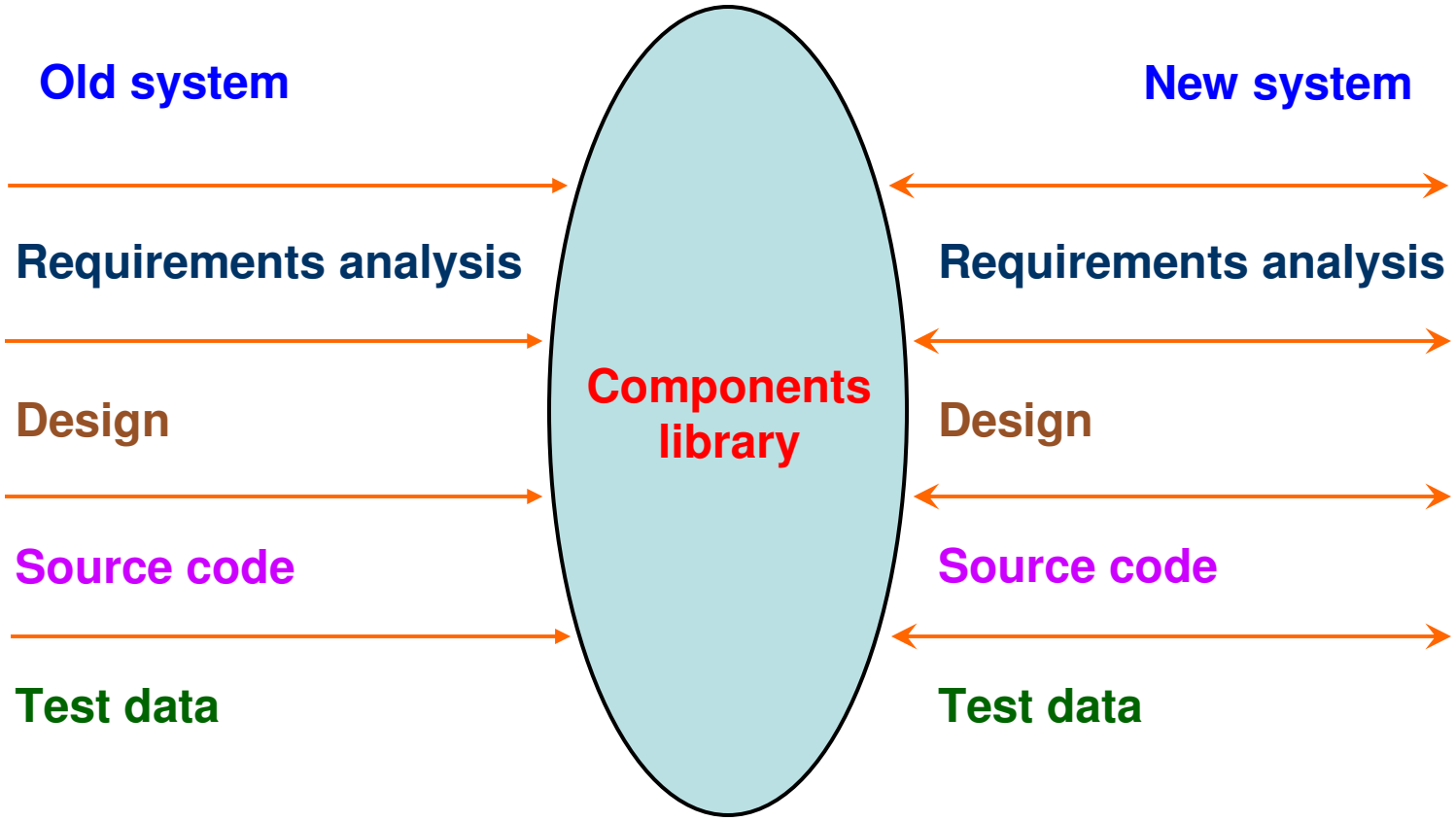


Fig. 5: The reuse model

Software Maintenance

- Boehm's Model

Boehm proposed a model for the maintenance process based upon the economic models and principles.

Boehm represent the maintenance process as a closed loop cycle.

Software Maintenance

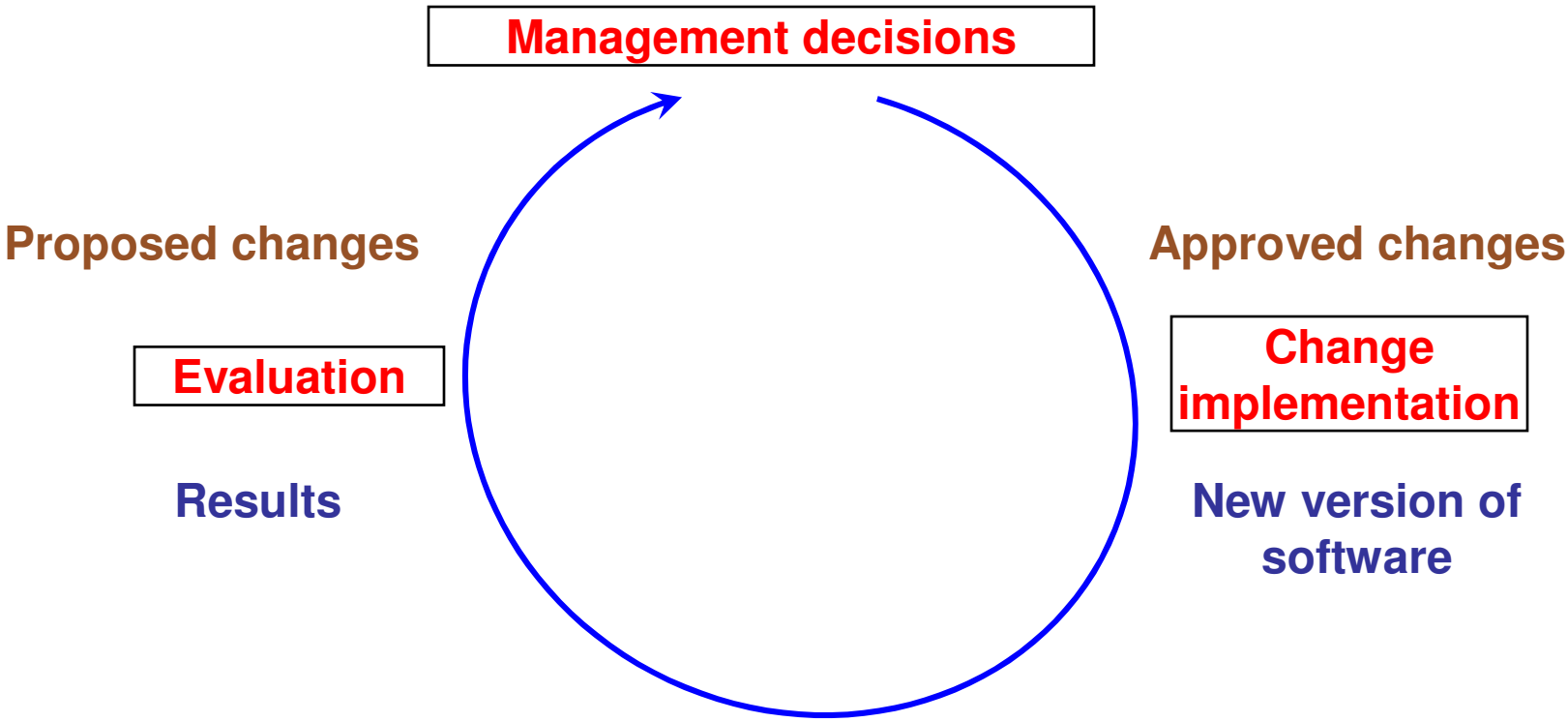


Fig. 6: Boehm's model

Software Maintenance

- Taute Maintenance Model

It is a typical maintenance model and has eight phases in cycle fashion. The phases are shown in Fig. 7

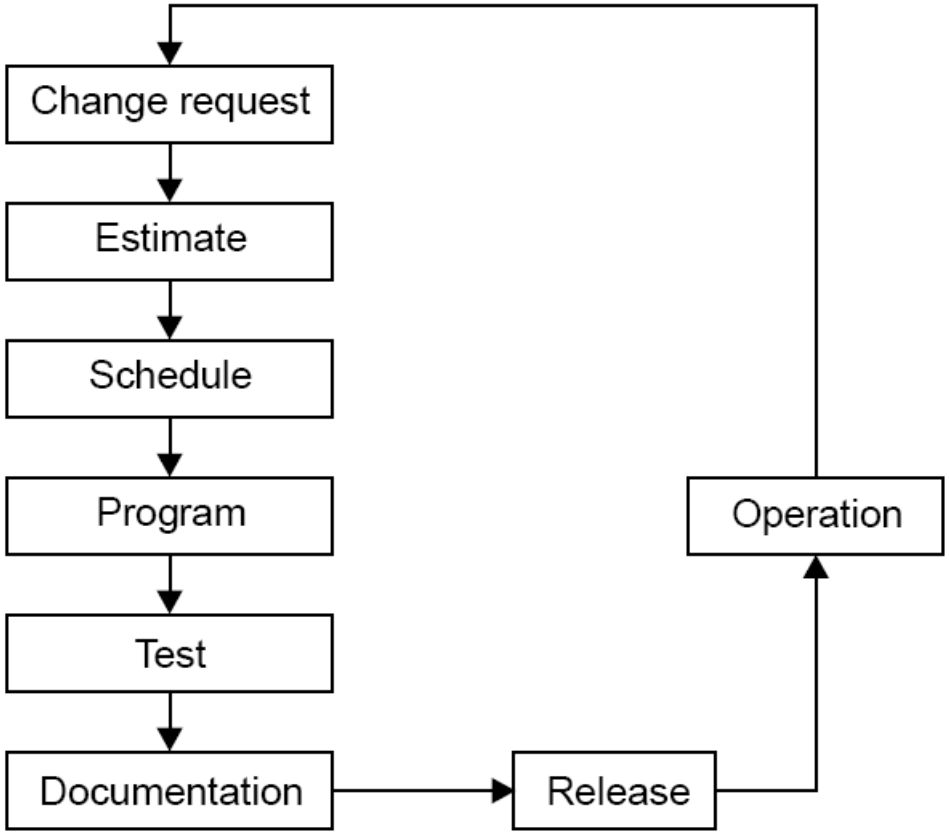


Fig. 7: Taute maintenance model

Software Maintenance

Phases :

1. Change request phase
2. Estimate phase
3. Schedule phase
4. Programming phase
5. Test phase
6. Documentation phase
7. Release phase
8. Operation phase

Estimation of maintenance costs

Phase	Ratio
Analysis	1
Design	10
Implementation	100

Table 3: Defect repair ratio

Software Maintenance

- Belady and Lehman Model

$$M = P + Ke^{(c-d)}$$

where

M : Total effort expended

P : Productive effort that involves analysis, design, coding, testing and evaluation.

K : An empirically determined constant.

c : Complexity measure due to lack of good design and documentation.

d : Degree to which maintenance team is familiar with the software.

Software Maintenance

Example – 9.1

The development effort for a software project is 500 person months. The empirically determined constant (K) is 0.3. The complexity of the code is quite high and is equal to 8. Calculate the total effort expended (M) if

(i) maintenance team has good level of understanding of the project (d=0.9)

(ii) maintenance team has poor understanding of the project (d=0.1)

Software Maintenance

Solution

Development effort (P) = 500 PM

K = 0.3

C = 8

(i) maintenance team has good level of understanding of the project (d=0.9)

$$M = P + Ke^{(c-d)}$$

$$= 500 + 0.3e^{(8-0.9)}$$

$$= 500 + 363.59 = 863.59 \text{ PM}$$

(ii) maintenance team has poor understanding of the project (d=0.1)

$$M = P + Ke^{(c-d)}$$

$$= 500 + 0.3e^{(8-0.1)}$$

$$= 500 + 809.18 = 1309.18 \text{ PM}$$

Software Maintenance

- Boehm Model

Boehm used a quantity called Annual Change Traffic (ACT).

"The fraction of a software product's source instructions which undergo change during a year either through addition, deletion or modification".

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

AME = ACT x SDE

Where, **SDE :** Software development effort in person months

ACT : Annual change Traffic

EAF : Effort Adjustment Factor

AME = ACT * SDE * EAF

Software Maintenance

Example – 9.2

Annual Change Traffic (ACT) for a software system is 15% per year. The development effort is 600 PMs. Compute estimate for Annual Maintenance Effort (AME). If life time of the project is 10 years, what is the total effort of the project ?

Software Maintenance

Solution

The development effort = 600 PM

Annual Change Traffic (ACT) = 15%

Total duration for which effort is to be calculated = 10 years

The maintenance effort is a fraction of development effort and is assumed to be constant.

$$AME = ACT \times SDE$$

$$= 0.15 \times 600 = 90 \text{ PM}$$

Maintenance effort for 10 years = 10 x 90 = 90 PM

Total effort = 600 + 900 = 1500 PM

Software Maintenance

Example – 9.3

A software project has development effort of 500 PM. It is assumed that 10% code will be modified per year. Some of the cost multipliers are given as:

1. Required software Reliability (RELY) : high
2. Date base size (DATA) : high
3. Analyst capability (ACAP) : high
4. Application experience (AEXP) : Very high
5. Programming language experience (LEXP) : high

Other multipliers are nominal. Calculate the Annual Maintenance Effort (AME).

Solution

Annual change traffic (ACT) = 10%

Software development effort (SDE) = 500 Pm

Using Table 5 of COCOMO model, effort adjustment factor can be calculated given below :

RELY = 1.15

ACAP = 0.86

AEXP = 0.82

LEXP = 0.95

DATA = 1.08

Software Maintenance

Other values are nominal values. Hence,

EAF = 1.15 x 0.86 x 0.82 x 0.95 x 1.08 = 0.832

AME = ACT * SDE * EAF

= 0.1 * 500 * 0.832 = 41.6 PM

AME = 41.6 PM

Regression Testing

Regression testing is the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously test code.

"Regression testing tests both the modified code and other parts of the program that may be affected by the program change. It serves many purposes :

- increase confidence in the correctness of the modified program
- locate errors in the modified program
- preserve the quality and reliability of software
- ensure the software's continued operation

Software Maintenance

- Development Testing Versus Regression Testing

Sr. No.	Development testing	Regression testing
1.	We create test suites and test plans	We can make use of existing test suite and test plans
2.	We test all software components	We retest affected components that have been modified by modifications.
3.	Budget gives time for testing	Budget often does not give time for regression testing.
4.	We perform testing just once on a software product	We perform regression testing many times over the life of the software product.
5.	Performed under the pressure of release date of the software	Performed in crisis situations, under greater time constraints.

Reverse Engineering

Reverse engineering is the process followed in order to find difficult, unknown and hidden information about a software system.

Software Maintenance

- Scope and Tasks

The areas there reverse engineering is applicable include (but not limited to):

1. Program comprehension
2. Redocumentation and/ or document generation
3. Recovery of design approach and design details at any level of abstraction
4. Identifying reusable components
5. Identifying components that need restructuring
6. Recovering business rules, and
7. Understanding high level system description

Software Maintenance

Reverse Engineering encompasses a wide array of tasks related to understanding and modifying software system. This array of tasks can be broken into a number of classes.

- Mapping between application and program domains

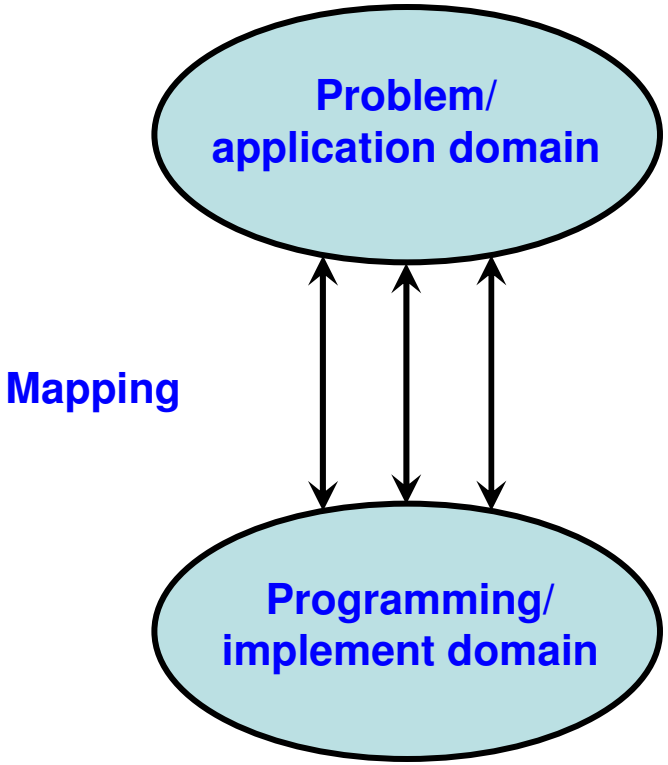


Fig. 10: Mapping between application and domains program

Software Maintenance

- Mapping between concrete and abstract levels
- Rediscovering high level structures
- Finding missing links between program syntax and semantics
- To extract reusable component

Software Maintenance

- Levels of Reverse Engineering

Reverse Engineers detect low level implementation constructs and replace them with their high level counterparts.

The process eventually results in an incremental formation of an overall architecture of the program.

Software Maintenance

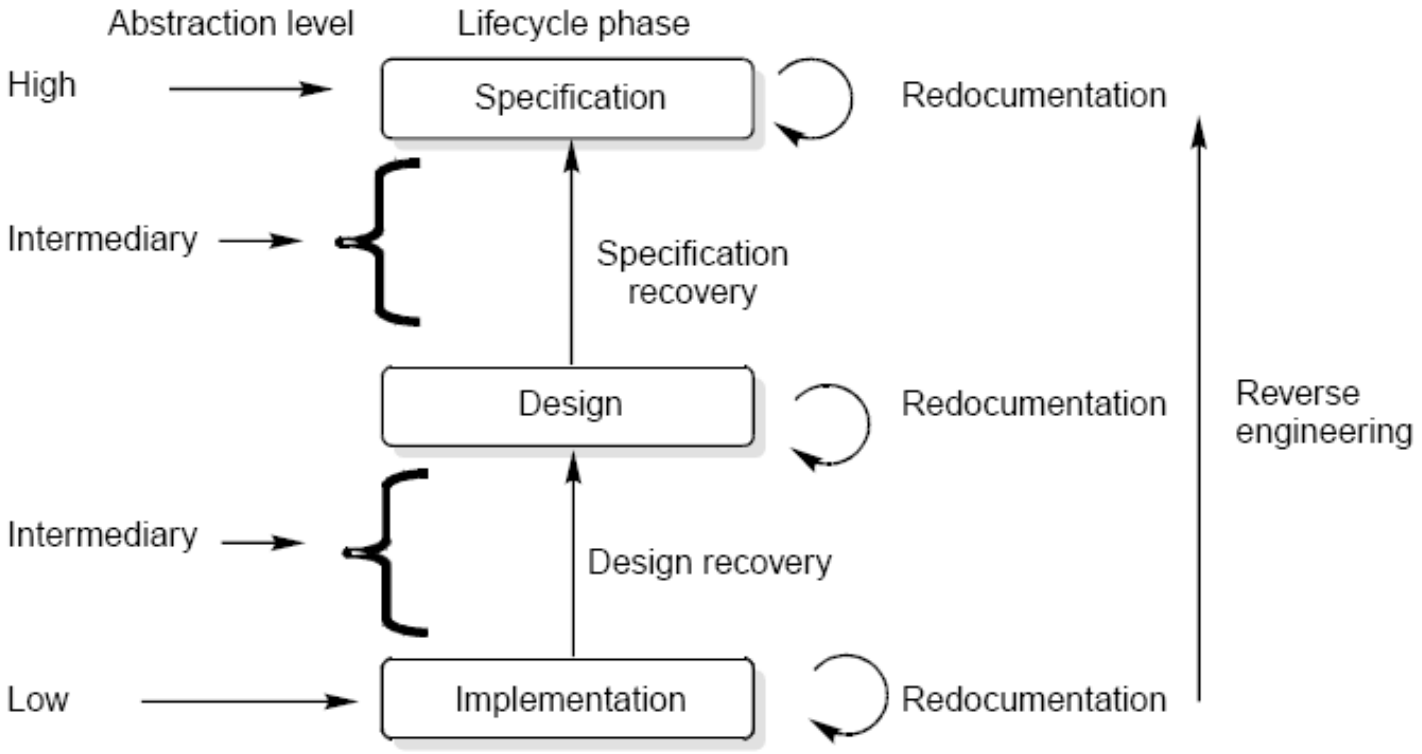


Fig. 11: Levels of abstraction

Software Maintenance

Redocumentation

Redocumentation is the recreation of a semantically equivalent representation within the same relative abstraction level.

Design recovery

Design recovery entails identifying and extracting meaningful higher level abstractions beyond those obtained directly from examination of the source code. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains.

Software RE-Engineering

Software re-engineering is concerned with taking existing legacy systems and re-implementing them to make them more maintainable.

The critical distinction between re-engineering and new software development is the starting point for the development as shown in Fig.12.

Software Maintenance

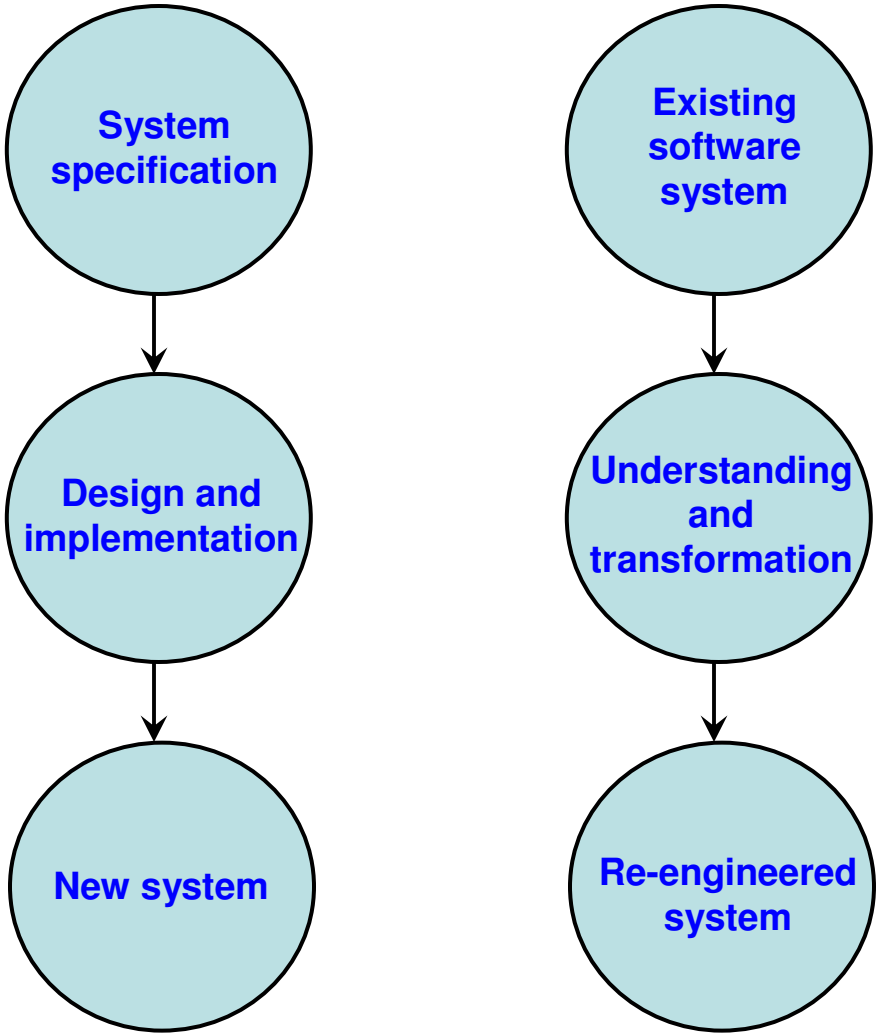


Fig. 12: Comparison of new software development with re-engineering

Software Maintenance

The following suggestions may be useful for the modification of the legacy code:

- ✓ Study code well before attempting changes
- ✓ Concentrate on overall control flow and not coding
- ✓ Heavily comment internal code
- ✓ Create Cross References
- ✓ Build Symbol tables
- ✓ Use own variables, constants and declarations to localize the effect
- ✓ Keep detailed maintenance document
- ✓ Use modern design techniques

Software Maintenance

- Source Code Translation

1. **Hardware platform update:** The organization may wish to change its standard hardware platform. Compilers for the original language may not be available on the new platform.

2. **Staff Skill Shortages:** There may be lack of trained maintenance staff for the original language. This is a particular problem where programs were written in some non standard language that has now gone out of general use.

3. **Organizational policy changes:** An organization may decide to standardize on a particular language to minimize its support software costs. Maintaining many versions of old compilers can be very expensive.

Software Maintenance

- Program Restructuring

1. **Control flow driven restructuring:** This involves the imposition of a clear control structure within the source code and can be either inter modular or intra modular in nature.

2. **Efficiency driven restructuring:** This involves restructuring a function or algorithm to make it more efficient. A simple example is the replacement of an IF-THEN-ELSE-IF-ELSE construct with a CASE construct.

Software Maintenance

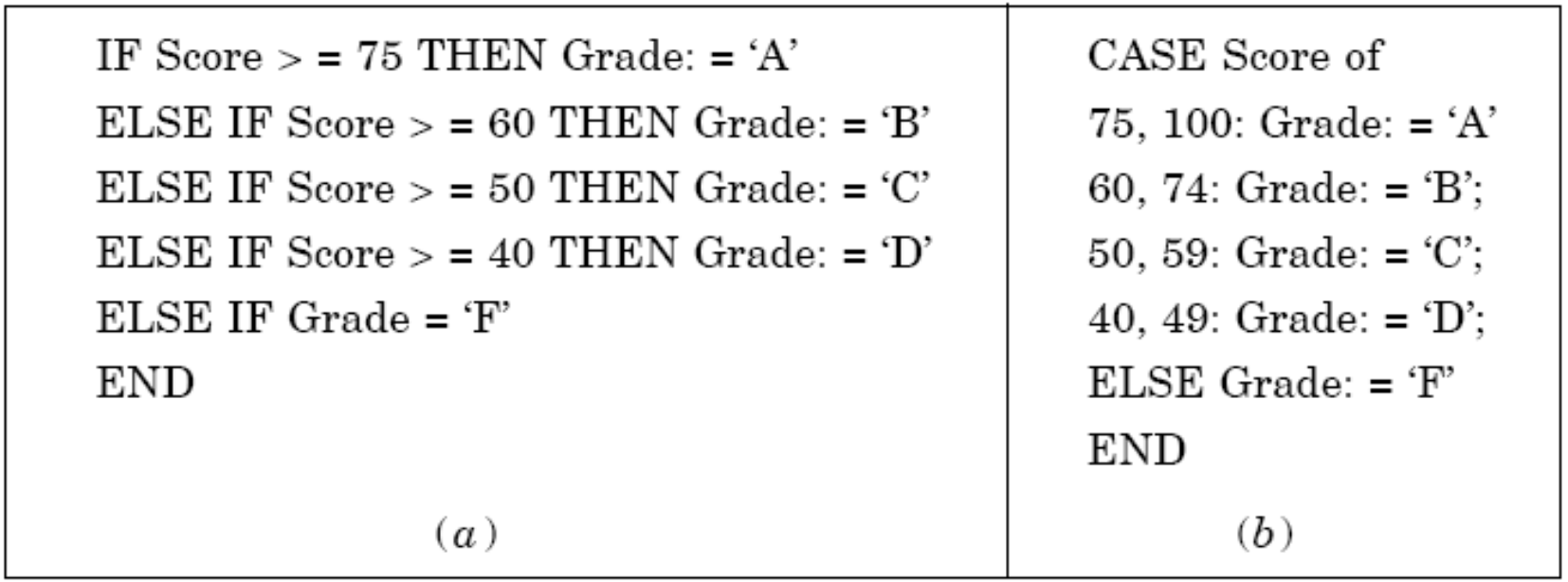


Fig. 13: Restructuring a program

Software Maintenance

3. **Adaption driven restructuring:** This involves changing the coding style in order to adapt the program to a new programming language or new operating environment, for instance changing an imperative program in PASCAL into a functional program in LISP.

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

9.1 Process of generating analysis and design documents is called

(a) Inverse Engineering
(b) Software Engineering
(c) Reverse Engineering
(d) Re-engineering

9.2 Regression testing is primarily related to

(a) Functional testing
(b) Data flow testing
(c) Development testing
(d) Maintenance testing

9.3 Which one is not a category of maintenance ?

(a) Corrective maintenance
(b) Effective maintenance
(c) Adaptive maintenance
(d) Perfective maintenance

9.4 The maintenance initiated by defects in the software is called

(a) Corrective maintenance
(b) Adaptive maintenance
(c) Perfective maintenance
(d) Preventive maintenance

9.5 Patch is known as

(a) Emergency fixes
(b) Routine fixes
(c) Critical fixes
(d) None of the above

Multiple Choice Questions

9.6 Adaptive maintenance is related to

(a) Modification in software due to failure

(b) Modification in software due to demand of new functionalities

(c) Modification in software due to increase in complexity

(d) Modification in software to match changes in the ever-changing environment.

9.7 Perfective maintenance refers to enhancements

(a) Making the product better

(b) Making the product faster and smaller

(c) Making the product with new functionalities

(d) All of the above

9.8 As per distribution of maintenance effort, which type of maintenance has consumed maximum share?

(a) Adaptive (b) Corrective

(c) Perfective (d) Preventive

9.9 As per distribution of maintenance effort, which type of maintenance has consumed minimum share?

(a) Adaptive (b) Corrective

(c) Perfective (d) Preventive

Multiple Choice Questions

9.10 Which one is not a maintenance model ?

(a) CMM
(b) Iterative Enhancement model
(c) Quick-fix model
(d) Reuse-Oriented model

9.11 In which model, fixes are done without detailed analysis of the long-term effects?

(a) Reuse oriented model
(b) Quick-fix model
(c) Taute maintenance model
(d) None of the above

9.12 Iterative enhancement model is a

(a) three stage model
(b) two stage model
(c) four stage model
(d) seven stage model

9.13 Taute maintenance model has

(a) Two phases
(b) six phases
(c) eight phases
(d) ten phases

9.14 In Boehm model, ACT stands for

(a) Actual change time
(b) Actual change traffic
(c) Annual change traffic
(d) Annual change time

Multiple Choice Questions

9.15 Regression testing is known as

(a) the process of retesting the modified parts of the software

(b) the process of testing the design documents

(c) the process of reviewing the SRS

(d) None of the above

9.16 The purpose of regression testing is to

(a) increase confidence in the correctness of the modified program

(b) locate errors in the modified program

(c) preserve the quantity and reliability of software

(d) All of the above

9.17 Regression testing is related to

(a) maintenance of software

(b) development of software

(c) both (a) and (b)

(d) none of the above.

9.18 Which one is not a selective retest technique

(a) coverage technique

(b) minimization technique

(c) safe technique

(d) maximization technique

Multiple Choice Questions

9.19 Purpose of reverse engineering is to

(a) recover information from the existing code or any other intermediate document

(b) redocumentation and/or document generation

(c) understand the source code and associated documents

(d) All of the above

9.20 Legacy systems are

(a) old systems

(b) new systems

(c) undeveloped systems

(d) None of the above

9.21 User documentation consists of

(a) System overview

(b) Installation guide

(c) Reference guide

(d) All of the above

9.22 Which one is not a user documentations ?

(a) Beginner's Guide

(b) Installation guide

(c) SRS

(d) System administration

Multiple Choice Questions

9.23 System documentation may not have

(a) SRS
(b) Design document
(c) Acceptance Test Plan
(d) System administration

9.24 The process by which existing processes and methods are replaced by new techniques is:

(a) Reverse engineering
(b) Business process re-engineering
(c) Software configuration management
(d) Technical feasibility

9.25 The process of transforming a model into source code is

(a) Reverse Engineering
(b) Forward engineering
(c) Re-engineering
(d) Restructuring

Exercises

9.1 What is software maintenance? Describe various categories of maintenance. Which category consumes maximum effort and why?

9.2 What are the implication of maintenance for a one person software production organisation?

9.3 Some people feel that "maintenance is manageable". What is your opinion about this issue?

9.4 Discuss various problems during maintenance. Describe some solutions to these problems.

9.5 Why do you think that the mistake is frequently made of considering software maintenance inferior to software development?

9.6 Explain the importance of maintenance. Which category consumes maximum effort and why?

9.7 Explain the steps of software maintenance with help of a diagram.

9.8 What is self descriptiveness of a program? Explain the effect of this parameter on maintenance activities.

Exercises

9.9 What is ripple effect? Discuss the various aspects of ripple effect and how does it affect the stability of a program?

9.10 What is maintainability? What is its role during maintenance?

9.11 Describe Quick-fix model. What are the advantage and disadvantage of this model?

9.12 How iterative enhancement model is helpful during maintenance? Explain the various stage cycles of this model.

9.13 Explain the Boehm's maintenance model with the help of a diagram.

9.14 State the various steps of reuse oriented model. Is it a recommended model in object oriented design?

9.15 Describe the Taute maintenance model. What are various phases of this model?

9.16 Write a short note on Boledy and Lehman model for the calculation of maintenance effort.

Exercises

9.17 Describe various maintenance cost estimation model.s

9.18 The development effort for a project is 600 PMs. The empirically determined constant (K) of Belady and Lehman model is 0.5. The complexity of code is quite high and is equal to 7. Calculate the total effort expended (M) if maintenance team has reasonable level of understanding of the project (d=0.7).

9.19 Annual change traffic (ACT) in a software system is 25% per year. The initial development cost was Rs. 20 lacs. Total life time for software is 10 years. What is the total cost of the software system?

9.20 What is regression testing? Differentiate between regression and development testing?

9.21 What is the importance of regression test selection? Discuss with help of examples.

9.22 What are selective retest techniques? How are they different from "retest-all" techniques?

Exercises

9.23 Explain the various categories of retest techniques. Which one is not useful and why?

9.24 What are the categories to evaluate regression test selection techniques? Why do we use such categorisation?

9.25 What is reverse engineering? Discuss levels of reverse engineering.

9.26 What are the appropriate reverse engineering tools? Discuss any two tools in detail.

9.27 Discuss reverse engineering and re-engineering.

9.28 What is re-engineering? Differentiate between re-engineering and new development.

9.29 Discuss the suggestions that may be useful for the modification of the legacy code.

9.30 Explain various types of restructuring techniques. How does restructuring help in maintaining a program?

Exercises

9.31 Explain why single entry, single exit modules make testing easier during maintenance.

9.32 What are configuration management activities? Draw the performa of change request form.

9.33 Explain why the success of a system depends heavily on the quantity of the documentation generated during system development.

9.34 What is an appropriate set of tools and documents required to maintain large software product/

9.35 Explain why a high degree of coupling among modules can make maintenance very difficult.

9.36 Is it feasible to specify maintainability in the SRS? If yes, how would we specify it?

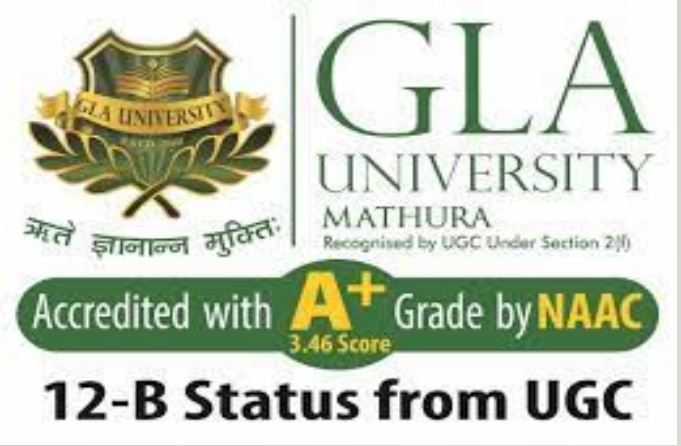
9.37 What tools and techniques are available for software maintenance? Discuss any two of them.

Exercises

9.38 Why is maintenance programming becoming more challenging than new development? What are desirable characteristics of a maintenance programmer?

9.39 Why little attention is paid to maintainability during design phase?

9.40 List out system documentation and also explain their purpose.



INTRODUCTION TO AUTOMATION TESTING AND TESTING TOOLS

MS. CHESHTAA BHARDWAJ

ASSISTANT PROFESSOR, DEPT. OF CEA,

GLA UNIVERSITY, MATHURA

Module No.	Content	Teaching Hours
I	**Introduction:** Introduction to Software Engineering, Software characteristics, Software Crisis, Software Engineering Process. **Software Development Life Cycle (SDLC) Models:** Waterfall, Incremental, Iterative Enhancement, Prototype, RAD and Spiral Models. **Software Requirements Engineering:** Types of Requirements, Requirement Elicitation Techniques Like Interviews, FAST & QFD, Use case Approach, Requirements Analysis Using DFD, Data Dictionaries & ER Diagrams, Requirements Documentation, and SRS. **Software Project Planning:** Size Estimation like Lines of Code & Function Count, Cost. **Estimation Models:** COCOMO (Basic, Intermediate) **Software Design:** Cohesion & Coupling, Classification of Cohesion & Coupling, Function Oriented Design, Object Oriented Design, Structure chart. **Coding:** Characteristics of Coding and Coding style.	20
II	**Software Metrics:** Software Measurements, Token Count, Halstead Software, Measures. **Software Reliability & Quality:** Introduction of Mc Call's & Boehm's Quality Model, Capability Maturity Models **Software Reliability Models**: Basic Execution Time Model. **Software Testing:** **Testing Fundamentals:** Test Case Design, Black Box Testing Strategies, White Box Testing, Unit Testing, Integration Testing, System Testing. **Introduction to Automation Testing and Testing Tools:** Automated Testing Process, Framework for Automation Testing, Introduction to Automation Testing Tool. **Software Maintenance:** Maintenance Process **Maintenance models**: Belady and Lehman Model, Boehm Model Regression Testing, Software Configuration Management; Implementation, Introduction to Reengineering and Reverse Engineering. **Software Risk Management:** Risk Identification and Risk Analysis	20

AUTOMATION TESTING

- Automated testing refers to using specialized software tools to control the execution of tests, compare actual outcomes with predicted outcomes, and generate reports automatically.
- It is an Automation Process of a Manual Process. It allows for executing repetitive tasks without the intervention of a Manual Tester.
- Automation tests can enter test data compare the expected result with the actual result and generate detailed test reports.
- The goal of automation tests is to reduce the number of test cases to be executed manually but not to eliminate manual testing.

WHY TRANSFORM FROM MANUAL TO AUTOMATED TESTING?

- Quality Assurance
- Error or Bug-free Software
- No Human Intervention
- Increased test coverage
- Testing can be done frequently

AUTOMATION TESTING PROCESS

- Test Tool Selection
- Define Scope of Automation
- Planning, Design, and Development
- Test Execution
- Maintenance (Creation of Reports)

FRAMEWORKS FOR AUTOMATION TESTING

- Data-Driven Framework
- Keyword-Driven Framework
- Modular Framework
- Hybrid Framework
- Behaviour-Driven Development (BDD) Framework
- Test-Driven Development (TDD) Framework
- Linear (or Scripted) Framework
- Continuous Integration (CI) Testing Framework…….& many more.

DATA-DRIVEN FRAMEWORK

- A data-driven framework runs test scripts with multiple data sets, stored externally, to ensure the application works with different inputs.
- It enables testing with multiple data sets without duplicating scripts, ideal for validating different inputs and large datasets.
- Example use case: Form submission, login scenarios, input validation.

KEYWORD-DRIVEN FRAMEWORK

- This framework breaks test cases into keywords (e.g., "click", "type", "verify") and data, with keywords listed in a table (often Excel), each linked to a specific action in the script.
- It enables testers with minimal programming skills to write and manage tests by defining keywords and test data, making it more accessible for non-technical users.
- Example use case: Functional testing, UI validations, form input actions.

MODULAR FRAMEWORK

- A modular framework divides the application into smaller components, tests them independently, and creates reusable functions for common actions, which can be used across different test scripts.
- The modular approach ensures reusability, reduces redundancy, and improves maintainability by allowing changes to be made in one place rather than updating all tests.
- Example use case: Testing login, search, and checkout functionality in an e-commerce application, where each module is tested independently.

HYBRID FRAMEWORK

- A hybrid framework combines elements from data-driven, keyword-driven, and modular frameworks etc., offering a flexible and comprehensive testing solution to meet various testing needs.
- Hybrid frameworks offer flexibility by integrating multiple testing methodologies in one framework, making them ideal for handling diverse testing needs in large-scale projects.
- Example use case: Large applications with both data-driven and keyword-driven tests, such as when testing both UI and backend services with different data sets and scenarios.

INTRODUCTION TO AUTOMATION TESTING TOOLS

- **Selenium** (web automation)
- **TestNG** (test framework)
- **Appium** (mobile app automation)
- **Cucumber** (BDD for collaboration)
- **Jenkins** (CI/CD integration)
- **PyTest** (Python testing)

……and many more.

SELENIUM (WEB AUTOMATION)

- **Characteristics:** Open-source tool to automate web browsers, supporting multiple languages (Java, Python, etc.).
- **When to Use:** For automating functional testing of web applications across different browsers.
- **Developed By:** Jason Huggins in 2004; maintained by the Selenium community

TestNG(TEST FRAMEWORK)

- **Characteristics:** Java-based testing framework with support for annotations, parallel execution, and reporting.
- **When to Use:** For unit and integration testing, especially in Java-based projects.
- **Developed By:** Cédric Beust in 2004.

APPIUM (MOBILE APP AUTOMATION)

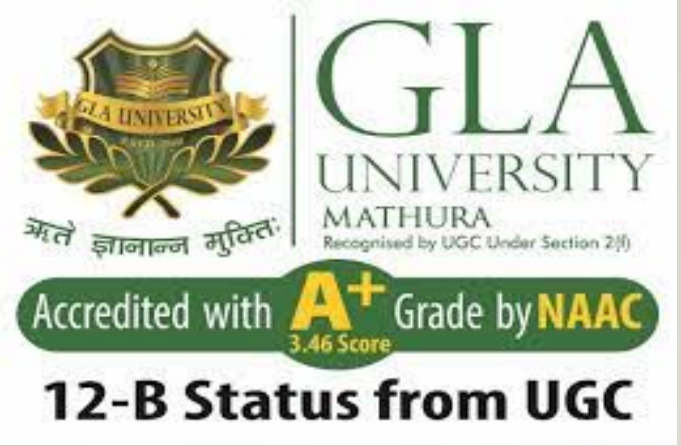
- **Characteristics:** Open-source tool for automating mobile apps (Android & iOS), supporting native and hybrid apps.
- **When to Use:** For automating functional testing of mobile applications.
- **Developed By:** Jonathan Lipps; maintained by the Appium community.

PYTEST (PYTHON TESTING)

- **Characteristics:** Python-based testing framework supporting unit, integration, and functional testing with fixtures and plugins.
- **When to Use:** For automated testing in Python-based projects.
- **Developed By:** Holger Krekel and maintained by the PyTest community.

JENKINS (CI/CD INTEGRATION)

- **Characteristics:** Open-source automation server for continuous integration (CI) and continuous delivery (CD).
- **When to Use:** For automating build, testing, and deployment pipelines in DevOps environments.
- **Developed By:** Kohsuke Kawaguchi in 2004.



SOFTWARE RISK MANAGEMENT

MS. CHESHTAA BHARDWAJ

ASSISTANT PROFESSOR, DEPT. OF CEA,

GLA UNIVERSITY, MATHURA

Module No.	Content	Teaching Hours
I	**Introduction:** Introduction to Software Engineering, Software characteristics, Software Crisis, Software Engineering Process. **Software Development Life Cycle (SDLC) Models:** Waterfall, Incremental, Iterative Enhancement, Prototype, RAD and Spiral Models. **Software Requirements Engineering:** Types of Requirements, Requirement Elicitation Techniques Like Interviews, FAST & QFD, Use case Approach, Requirements Analysis Using DFD, Data Dictionaries & ER Diagrams, Requirements Documentation, and SRS. **Software Project Planning:** Size Estimation like Lines of Code & Function Count, Cost. **Estimation Models:** COCOMO (Basic, Intermediate) **Software Design:** Cohesion & Coupling, Classification of Cohesion & Coupling, Function Oriented Design, Object Oriented Design, Structure chart. **Coding:** Characteristics of Coding and Coding style.	20
II	**Software Metrics:** Software Measurements, Token Count, Halstead Software, Measures. **Software Reliability & Quality:** Introduction of Mc Call's & Boehm's Quality Model, Capability Maturity Models **Software Reliability Models**: Basic Execution Time Model. **Software Testing:** **Testing Fundamentals:** Test Case Design, Black Box Testing Strategies, White Box Testing, Unit Testing, Integration Testing, System Testing. **Introduction to Automation Testing and Testing Tools:** Automated Testing Process, Framework for Automation Testing, Introduction to Automation Testing Tool. **Software Maintenance:** Maintenance Process **Maintenance models**: Belady and Lehman Model, Boehm Model Regression Testing, Software Configuration Management; Implementation, Introduction to Reengineering and Reverse Engineering. **Software Risk Management:** Risk Identification and Risk Analysis	20

WHAT IS RISK?

- "Tomorrow problems are today's risk."
- A "risk" is a problem that could cause some loss or threaten the progress of the project, but which has not happened yet.
- These potential issues might harm cost, schedule or technical success of the project and the quality of our software device, or project team morale.
- Risk Management is the system of identifying addressing and eliminating these problems before they can damage the project.
- We need to differentiate risks, as potential issues, from the current problems of the project.
- Different methods are required to address these two kinds of issues.

RISK MANAGEMENT

Risk management is the process of identifying addressing and eliminating these problems(Current problems & Potential Problems) before they can damage the project.

TYPICAL SOFTWARE RISK

Capers Jones has identified the top five risk factors that threaten projects in different applications.

1. **Dependencies on outside agencies or factors.**
    - Availability of trained, experienced persons
    - Inter group dependencies
    - Customer-Furnished items or information
    - Internal & external subcontractor relationships

2. Requirement Issues

- Uncertain Requirements
- Lack of clear product vision
- Lack of agreement on product requirements
- Unprioritized requirements
- New market with uncertain needs
- Rapidly changing requirements
- Inadequate Impact analysis of requirements changes
- Wrong Product/Right product but have some faults (Either situation results in unpleasant surprises and unhappy customers.)

3. Management Issues

Project managers usually write the risk management plans, and most people do not wish to air their weaknesses in public.

- Inadequate planning
- Inadequate visibility into actual project status
- Unclear project ownership and decision making
- Staff personality conflicts
- Unrealistic expectation
- Poor communication

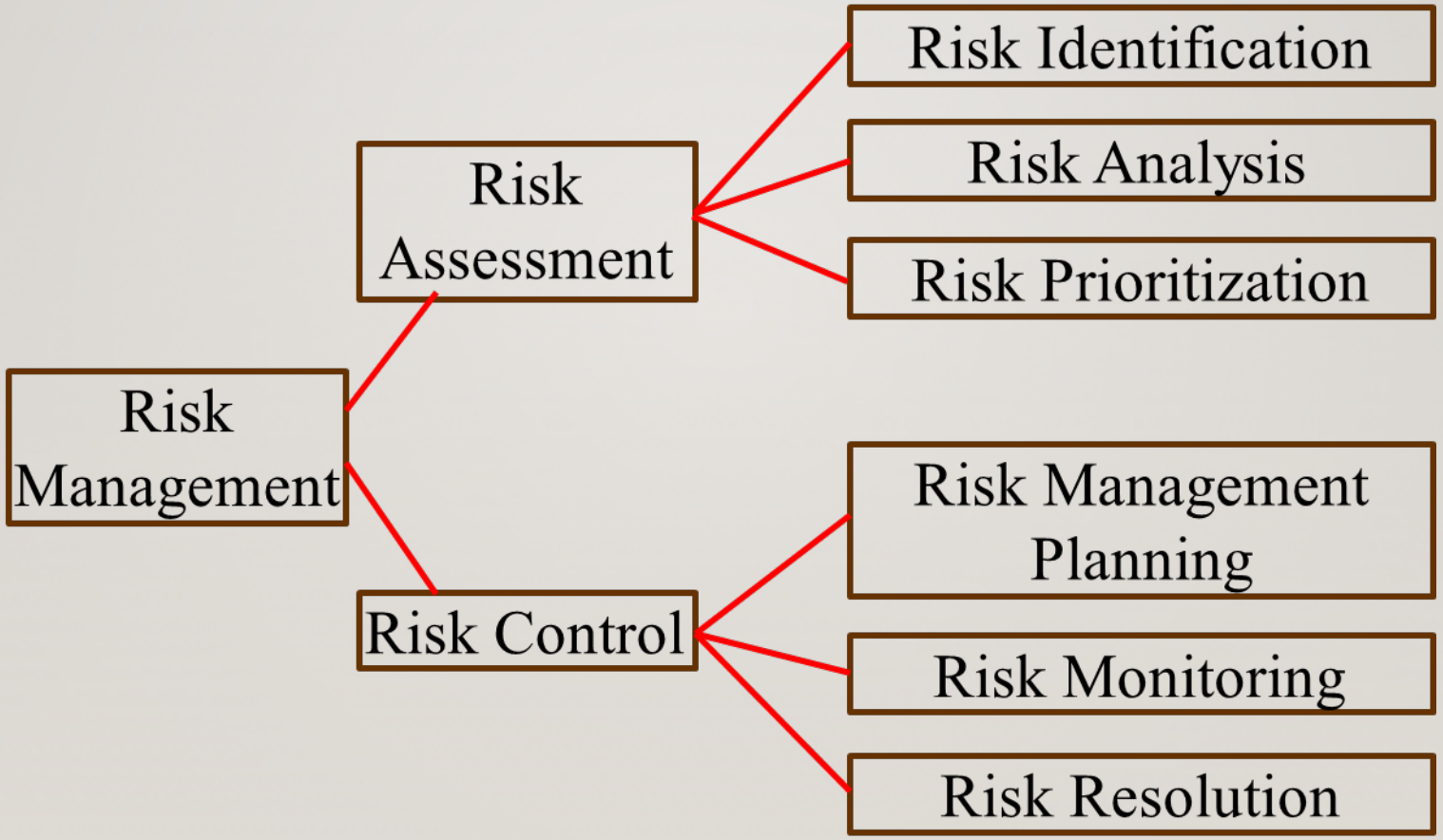
4. Lack of knowledge

- Inadequate training
- Poor understanding of methods, tools, and techniques
- Inadequate application domain experience
- New Technologies
- Ineffective, poorly processes documented or neglected

5. Other risk categories

- Unavailability of adequate testing facilities
- Turnover of essential personnel
- Unachievable performance requirements
- Technical approaches that may not work

RISK MANAGEMENT ACTIVITIES



RISK ASSESSMENT (IDENTIFICATION OF RISKS)

- **Risk analysis** involves examining how project outcomes might change with modification of risk input variables.
- **Risk prioritization** focus for severe risks.
- **Risk exposure:** It is the product of the probability of incurring a loss due to the risk and the potential magnitude of that loss.

Another way of handling risk is the risk avoidance. Do not do the risky things! We may avoid risks by not undertaking certain projects, or by relying on proven rather than cutting edge technologies.

RISK CONTROL

Risk Management Planning produces a plan for dealing with each significant risks.

- Record decision in the plan.

Risk resolution is the execution of the plans of dealing with each risk.

E-BOOKS

- **Software Engineering- A Practitioner's Approach** by **Roger S. Pressman**
- **Software Engineering** Ninth Edition by **Ian Sommerville**
- **Software Engineering** Third Edition by **K.K Aggarwal & Yogesh Singh**